

普通高等教育“十一五”国家级规划教材  
华南理工大学精品课程教材  
高等学校计算机基础及应用教材

# 数据结构与算法

## （C++语言版）

肖南峰 赵 洁 等编

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

## 内 容 简 介

本书为普通高等教育“十一五”国家级规划教材。

全书共分 15 章，主要内容包括：绪论、线性表、栈和队列、串、多维数组和广义表、树和二叉树、图、查找、内部排序、文件组织和外排序、贪婪算法、分而治之算法、动态规划、回溯、分枝定界法。在前 10 章中，对相应的数据结构的 ADT 描述、存储结构、基本操作、综合算法做了全面深入的阐述，每章的最后都对该章的基本内容、学习要点、具体要求、重点和难点进行了归纳和总结。在第 11~15 章中，列举了几个应用多种数据结构进行综合性算法设计的典型例子。另外，作者在参考了近年来许多的国内外教材之后，选编了大量精心设计的习题。本书每章的学习内容翔实，算法和例题典型，而且给出了对应的 VC++ 6.0 源程序。本书免费提供电子课件。

本书不仅可作为计算机学科各专业学生的教材，也适合作为广大工程技术人员和自学考试人员的参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

### 图书在版编目（CIP）数据

数据结构与算法：C++语言版/肖南峰，赵洁等编. —北京：电子工业出版社，2009.5

高等学校计算机基础及应用教材

ISBN 978-7-121-08301-3

I. 数… II. ①肖…②赵… III. ①数据结构—高等学校—教材②算法分析—高等学校—教材③C 语言—程序设计—高等学校—教材 IV. TP311.12 TP312

中国版本图书馆 CIP 数据核字（2009）第 021494 号

责任编辑：冉 哲

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×1092 1/16 印张：19.75 字数：488 千字

印 次：2009 年 5 月第 1 次印刷

印 数：2009 册 定价：29.80 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系及邮购电话：（010）88254888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线：（010）88258888。

# 前 言

“数据结构”是计算机学科各个专业的一门重要的专业基础课程。本课程主要讲授数据的逻辑结构、存储结构、基本运算、运算实现、算法设计、算法分析、算法评价等方面的内容，使学生对线性表、栈、队列、串、数组、树及二叉树、无向图和有向图、静态及动态查找表、文件等各种数据结构有深刻的理解，对各种常见的排序方法与算法有深入的了解。在此基础上，还要求学生系统地掌握在不同的存储结构上利用上述数据结构进行综合性算法设计的方法和技巧。因此，它是一门理论性和实践性都很强的课程。

根据作者多年的教学实践发现，学生对于数据结构的应用，特别是在做算法设计习题和编程上机实习这两个环节上都不同程度地存在着一定的困难。为了帮助学生更好地掌握该课程的知识，提高算法设计和动手编程的能力，急需为计算机学科各专业开设的“数据结构”课程编写一本基础扎实、知识面广、适应性强的教材。为此，在华南理工大学精品课程建设基金的资助下，我们编写了这本《数据结构与算法（C++语言版）》教材，主要目的就是加强基础、拓宽知识面、增强适应性，以便使学生能够更深入地理解教材内容，开拓思想，培养并掌握良好的算法设计与程序实现的技能，以及解决实际问题的能力。

本书为普通高等教育“十一五”国家级规划教材。

本书共分 15 章，主要内容包括：绪论、线性表、栈和队列、串、多维数组和广义表、树和二叉树、图、查找、内部排序、文件组织和外排序、贪婪算法、分而治之算法、动态规划、回溯、分枝定界法。在前 10 章中，对相应的数据结构的 ADT 描述、存储结构、基本操作、综合算法做了全面深入的阐述，每章的最后都对该章的基本内容、学习要点、具体要求、重点难点进行归纳和总结。在第 11~15 章中，列举了几个应用多种数据结构进行综合性算法设计的典型例子。另外，作者在参考了近年来许多的国内外教材之后，选编了大量精心设计的习题。本书每章的学习内容翔实，算法和例题典型，而且给出了对应的 VC++ 6.0 源程序。本书提供免费电子课件。

本书不仅可作为计算机学科各专业学生的教材，也适合作为广工程技术人员和自学考试人员的参考书。

肖南峰教授、黄敏讲师和张琴讲师编写了第 1~10 章并选编了全部习题，赵洁讲师编写了第 11~15 章及所有的 Visual C++ 6.0 源程序，吕建明讲师校对了部分章节的习题。在本教材的编写过程中，华南理工大学“数据结构”精品课程课题组和“智能计算机科研团队”的多位教师提出了许多的宝贵意见，我们在此向他们表示衷心的感谢。另外，还要感谢华南理工大学精品课程建设基金的支持。由于作者水平有限，教材中难免会存在错误，因此热忱地欢迎广大读者提出批评和意见。

编 者

2009 年 3 月

于华南理工大学

## 作者简介



肖南峰博士，男，1962 年 11 月生，华南理工大学计算机科学与工程学院教授，博士生导师。1982 年 7 月毕业于华中工学院（现为华中科技大学）自动控制与计算机工程系，获工学学士学位；1989 年 1 月毕业于东北工学院（现为东北大学），获工学硕士学位；2001 年 6 月毕业于日本横浜国立大学，获工学博士学位。2001 年 9 月至 2002 年 9 月在澳大利亚 Deakin 大学从事科学研究。他作为主持人先后完成了 2 项国家自然科学基金项目、2 项广东省自然科学基金重点项目，1 项教育部留学回国人员科研启动基金项目，以及由广东省教育厅和华南理工大学等资助的 20 多项教学与科研课题，在国内外发表学术论文 120 多篇，其中被三大索引收录近 50 篇，出版专著和教材 5 部，申请或获得发明及实用新型专利 5 项，软件版权 10 项。他一直在教学一线从事“数据结构”等课程的教学，已先后为近 20 届计算机专业、计算机辅修专业、电类联合班、继续教育学院和网络学院的本科生讲授过“数据结构”、“高级程序设计语言”等课程，积累了丰富的教学经验。

# 目 录

第 1 章 绪论	1
1.1 什么是数据结构	1
1.1.1 基本概念	1
1.1.2 数据结构的内涵	1
1.1.3 数据类型和抽象数据类型	3
1.2 算法和算法分析	4
1.2.1 算法的描述	4
1.2.2 算法设计的要求	4
1.2.3 算法分析	4
本章总结	7
习题 1	8
第 2 章 线性表	12
2.1 线性表的类型定义	12
2.1.1 基本概念	12
2.1.2 抽象数据类型描述	12
2.1.3 线性表抽象类	13
2.1.4 异常类 NoMem 和 OutOfBounds	14
2.2 线性表的顺序存储结构	15
2.2.1 基本概念	15
2.2.2 基本操作	16
2.3 线性表的链式存储结构	21
2.3.1 线性链表	21
2.3.2 循环链表	29
2.3.3 双向链表	29
2.3.4 顺序表和链表的比较	32
2.4 线性表的应用——多项式相加与 Josephus 问题	32
2.4.1 多项式表示	32
2.4.2 多项式相加	35
本章总结	37
习题 2	38
第 3 章 栈与队列	43
3.1 栈	43
3.1.1 栈的定义	43
3.1.2 栈的抽象类	44
3.1.3 栈的顺序存储结构	44
3.1.4 栈的链式存储结构	46

3.2	栈的应用举例	47
3.3	栈与递归	51
3.4	队列	51
3.4.1	队列的定义	51
3.4.2	队列的顺序存储结构	53
3.4.3	队列的链式存储结构	58
	本章总结	60
	习题 3	61
<b>第 4 章</b>	<b>串</b>	<b>64</b>
4.1	串的逻辑结构	64
4.1.1	基本概念	64
4.1.2	串的大小比较	66
4.2	串的存储结构	66
4.3	串函数与串的分类	67
4.3.1	常用的 C++ 串函数	67
4.3.2	串的分类	68
4.4	串模式匹配	70
4.4.1	简单串模式匹配算法	71
4.4.2	无回溯的匹配算法	71
4.5	串的应用——文本编辑	73
	本章总结	74
	习题 4	74
<b>第 5 章</b>	<b>多维数组与广义表</b>	<b>76</b>
5.1	数组	76
5.1.1	数组的定义	76
5.1.2	C++ 的数组	77
5.1.3	数组的存储结构与寻址问题	77
5.2	类 Array1D	80
5.3	矩阵的压缩存储	82
5.3.1	特殊矩阵	83
5.3.2	稀疏矩阵	85
5.4	十字链表	89
5.4.1	存储方式	89
5.4.2	十字链表对象	90
5.4.3	基本操作的实现	91
5.4.4	十字链表相加法*	93
5.5	广义表	95
5.5.1	广义表的定义	95
5.5.2	广义表的抽象数据类型定义	96
5.5.3	广义表的存储结构	97

本章总结	100
习题 5	101
<b>第 6 章 树与二叉树</b>	<b>103</b>
6.1 树的相关概念	103
6.1.1 树的递归定义和逻辑表示法	103
6.1.2 树的基本术语	103
6.1.3 树的抽象类型定义	104
6.2 树的存储结构与遍历	105
6.2.1 树的存储结构	105
6.2.2 树与森林的遍历	108
6.3 二叉树	109
6.3.1 二叉树的定义	109
6.3.2 二叉树的性质	111
6.4 二叉树的存储结构	112
6.4.1 顺序存储结构	112
6.4.2 链式存储结构	113
6.5 二叉树对象模型	114
6.5.1 二叉树结点对象	114
6.5.2 二叉树对象	115
6.6 二叉树的遍历与线索化	118
6.6.1 二叉树的遍历	118
6.6.2 二叉树的线索化	123
6.6.3 二叉树与森林的转换	126
6.7 哈夫曼树及其应用	128
6.7.1 哈夫曼树	128
6.7.2 哈夫曼编码	129
本章总结	133
习题 6	134
<b>第 7 章 图</b>	<b>137</b>
7.1 图的定义和术语	137
7.2 图的对象抽象模型	141
7.2.1 图结点对象抽象模型	141
7.2.2 图的边对象抽象模型	141
7.2.3 图对象抽象模型	142
7.3 图的存储结构	143
7.3.1 邻接矩阵	143
7.3.2 邻接表	148
7.3.3 十字链表(有向图)	153
7.3.4 邻接多重表(无向图)	155
7.4 图的遍历	156

7.4.1	深度优先遍历	156
7.4.2	广度优先遍历	161
7.5	图的连通性问题	163
7.5.1	图的连通分量	163
7.5.2	生成树及生成森林	164
7.6	有向无环图及其应用	167
7.6.1	有向无环图	167
7.6.2	AOV 网与拓扑排序	168
7.6.3	AOE 网与关键路径	171
	本章总结	176
	习题 7	176
<b>第 8 章</b>	<b>查找</b>	179
8.1	查找表的相关概念	179
8.1.1	基本概念	179
8.1.2	类型说明	179
8.2	静态查找表	180
8.2.1	概述	180
8.2.2	顺序表的查找	180
8.2.3	有序表的查找	182
8.2.4	索引顺序表的查找	183
8.3	动态查找表	186
8.3.1	概述	186
8.3.2	二叉排序树	186
8.3.3	平衡二叉树	191
8.3.4	B-树和 B <sup>+</sup> 树	193
8.4	哈希表	198
8.4.1	哈希表的定义	198
8.4.2	哈希函数的构造	198
8.4.3	处理冲突的方法	200
8.4.4	哈希表的查找及其分析	204
	本章总结	205
	习题 8	206
<b>第 9 章</b>	<b>内部排序</b>	208
9.1	排序的基本概念	208
9.2	插入排序	209
9.2.1	直接插入排序	209
9.2.2	折半插入排序	211
9.2.3	2 路插入排序	211
9.2.4	表插入排序	212
9.2.5	希尔排序	214



9.3	交换排序 .....	215
9.3.1	冒泡排序 .....	215
9.3.2	快速排序 .....	216
9.4	选择排序 .....	218
9.4.1	简单选择排序 .....	218
9.4.2	堆排序 .....	219
9.5	归并排序 .....	224
9.6	基数排序 .....	225
9.6.1	多关键码的排序 .....	225
9.6.2	链式基数排序 .....	226
9.7	内排序方法的比较和讨论 .....	227
	本章总结 .....	228
	习题 9 .....	229
<b>第 10 章</b>	<b>文件组织和外排序 .....</b>	<b>231</b>
10.1	外存储器概述 .....	231
10.1.1	磁带及其信息的存取 .....	231
10.1.2	磁盘及其信息的存取 .....	232
10.1.3	U 盘 .....	232
10.2	文件的基本概念 .....	233
10.2.1	文件 .....	233
10.2.2	文件的操作（运算）与存取 .....	233
10.2.3	文件的物理结构 .....	234
10.3	顺序文件 .....	235
10.4	索引文件 .....	235
10.5	ISAM 文件和 VSAM 文件 .....	236
10.5.1	ISAM 文件 .....	236
10.5.2	VSAM 文件 .....	238
10.6	散列文件 .....	239
10.7	多关键字文件 .....	240
10.7.1	多重表文件 .....	241
10.7.2	倒排文件 .....	241
10.8	外部排序 .....	242
	本章总结 .....	243
	习题 10 .....	244
<b>第 11 章</b>	<b>贪婪算法 .....</b>	<b>247</b>
11.1	最优化问题 .....	247
11.2	算法思想 .....	248
11.3	应用 .....	248
11.3.1	货箱装船 .....	248
11.3.2	0-1 背包问题 .....	249

11.3.3	拓扑排序	250
11.3.4	二分覆盖	250
11.3.5	单源最短路径	254
11.3.6	最小代价生成树	256
习题 11		259
第 12 章	分而治之算法	261
12.1	算法思想	261
12.2	应用	261
12.2.1	最大最小问题	261
12.2.2	归并排序	263
12.2.3	快速排序	265
12.2.4	选择问题	265
12.2.5	距离最近的点对问题	267
习题 12		271
第 13 章	动态规划	272
13.1	算法思想	272
13.2	应用	272
13.2.1	0-1 背包问题	272
13.2.2	图像压缩	274
13.2.3	矩阵连乘法	279
习题 13		282
第 14 章	回溯	284
14.1	算法思想	284
14.2	应用	285
14.2.1	货箱装船	285
14.2.2	0-1 背包问题	288
14.2.3	最大完备子图	291
习题 14		293
第 15 章	分枝定界法	294
15.1	算法思想	294
15.2	应用	295
15.2.1	货箱装船	295
15.2.2	0-1 背包问题	301
15.2.3	最大完备子图	302
习题 15		304
参考文献		305

# 第 1 章 绪 论

从世界上第一台计算机诞生至今，已有 60 多年的历史。在这期间，计算机的发展和应用已经渗透到了人类社会的各个领域，计算机加工和处理的对象也从纯粹的数值发展到了字符、图像、声音等各种具有一定结构的数据。为了更好地设计程序，以提高计算机在解决复杂问题时的处理效率，研究数据的特性和数据之间存在的关系至关重要。“数据结构”作为计算机科学与技术领域中的一门专业基础课，它专门研究数据的特性和数据之间存在的关系，以及如何在计算机中有效地存取数据和处理数据。因此，“数据结构”是设计和实现编译程序、操作系统、数据库系统和大型应用程序的重要基础，它也是介于数学、计算机硬件和计算机软件之间的一门核心课程，并将随着人类社会的各个领域计算问题的不断深入研究而继续发展。

## 1.1 什么是数据结构

### 1.1.1 基本概念

(1) 数据：信息的载体，是客观事物的符号表示。数据能够被计算机识别、存取和处理，数据也是计算机程序加工和处理的“原料”。例如，实数、字符串、图像和声音等。

(2) 数据项：具有独立含义的最小标识单位。例如，字段、域、属性等。

(3) 数据元素：数据的基本单位。一个数据元素可由若干个数据项组成。

(4) 数据对象：性质相同的数据元素的集合，是数据的一个子集。例如，26 个英文字母构成的字符集合，一个学校全体学生或教师构成的学生集合或教师集合等。

(5) 数据结构：相互之间存在一种或多种特定关系的数据元素的集合，即数据的组织形式。数据结构的形式化定义通常用一个二元组  $\text{Data\_Structure}=(D, R)$  来表示，式中， $D$  是数据元素的有限集（也即数据对象）， $R$  是  $D$  上关系的有限集。

### 1.1.2 数据结构的内涵

数据结构一般包含数据的逻辑结构和存储结构及数据运算。数据结构的研究内容包括两方面：一方面是抽象地研究数据集合的结构（抽象数据结构）特点；另一方面是研究如何把抽象数据结构转化为存储组织形式（数据的存储结构）。这两方面的研究既可独立于各个领域的应用来研究，也可结合具体应用领域中的特点来进行研究。例如，专门研究计算机图像识别或定理证明中的数据结构。目前，从抽象数据类型的观点来讨论数据结构已经成为一种新的趋势，并越来越被人们所重视。

### 1. 数据的逻辑结构

数据的逻辑结构是指数据元素以及它们相互之间的逻辑关系，数据的逻辑结构与数据的存储无关。根据数据元素之间关系的不同特性，通常有 4 类逻辑结构：① 集合，集合的

逻辑结构中所有数据元素都属于同一个集合，所有数据元素杂乱无章地聚集在一起，各个数据元素之间无任何联系；② 线性结构，逻辑结构中的数据元素之间存在着一个对一的关系，各个数据元素之间通常有严格的先后次序关系；③ 树形结构，逻辑结构中的数据元素之间存在着一个对多个的关系，各个数据元素之间通常有严格的层次关系；④ 图状结构，逻辑结构中的数据元素之间存在着多个对多个的关系，各个数据元素之间均可能存在相互联系。

在不产生混淆的前提下，常将数据的逻辑结构简称为数据结构。除了上述 4 类逻辑结构之外，根据数据元素（结点）之间的前后相邻关系，数据的逻辑结构还可分为线性结构和非线性结构两大类：① 线性结构的逻辑特征是，若结构是非空集，则有且仅有一个开始结点和一个终端结点，并所有结点都最多只有一个直接前驱结点和一个直接后继结点。线性表是一个典型的线性结构，栈、队列和串等都是线性结构；② 非线性结构的逻辑特征是，一个结点可能有多个直接前驱和直接后继。树和图都是非线性结构。

【例 1-1】 怎样描述数据的逻辑结构？

对数据元素之间关系的描述是数据的逻辑结构，它可形式地用一个二元组表示为  $K=(D, R)$ ，式中， $D$  是由有限个结点所构成的集合， $R$  是由有限个关系所构成的集合。有时为了直观起见，也用以图示法来表示数据的逻辑结构。逻辑结构与使用的计算机无关。例如，一批数据的逻辑结构  $K=(D, R)$ ，式中， $D=\{d_1, d_2, \dots, d_9\}$ ， $R=\{<d_1, d_2>, <d_1, d_3>, <d_1, d_4>, <d_1, d_7>, <d_1, d_8>, <d_4, d_5>, <d_4, d_6>, <d_8, d_9>\}$ ，则该批数据的逻辑结构如图 1.1 所示。

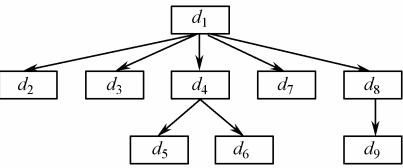


图 1.1 数据的逻辑结构示例

对于  $R$  中包含有多种关系的情况，也可用类似的方法描述。

2. 数据的存储结构

数据的存储结构（物理结构）是指数据在计算机中的存储表示，它包括数据元素的表示和关系的表示。数据的存储结构有以下 4 种基本存储方法。

① 顺序存储。该存储方法把逻辑上相邻的结点存储在物理位置上相邻的存储单元中，结点间的逻辑关系由存储单元的邻接关系来体现。由此得到的存储表示称为顺序存储结构（sequential storage structure）。该方法通常借助于高级程序语言中的数组来描述，且主要应用于线性的数据结构。非线性的数据结构也可通过线性化的方法来实现顺序存储。

② 链接存储。该方法不要求逻辑上相邻的结点在物理位置上也相邻，结点之间的逻辑关系由附加的指针字段表示。由此得到的存储表示称为链式存储结构（linked storage structure），通常借助于高级程序语言的指针类型来描述。

③ 索引存储。该方法通常在存储结点信息的同时，还要建立附加的索引表。索引表由若干索引项组成。若每个结点在索引表中都有一个索引项，则该索引表称之为稠密索引（dense index）。若一组结点在索引表中只对应一个索引项，则该索引表称为稀疏索引（sparse index）。索引项的形式一般是（关键字，地址），关键字是能唯一标识一个结点的那些数据项。稠密索引中索引项的地址指示结点所在的存储位置，稀疏索引中索引项的地址指示一组结点的起始存储位置。

④ 散列存储。该方法根据结点的关键字直接计算出该结点的存储地址。

以上述 4 种基本存储方法既可单独使用，也可组合起来对数据结构进行存储映射。同一种逻辑结构采用不同的存储方法，可得到不同的存储结构。选择何种存储结构来表示相应的逻辑结构，要视具体的应用要求而定，主要考虑运算方便和算法的时空效率要求。

### 3. 数据的运算

数据的运算是 对数据施加的操作。数据的运算定义在数据的逻辑结构上，每种逻辑结构都有一个运算的集合。在数据结构中，运算不仅仅是加、减、乘、除等运算，大多数的运算都涉及算法的实现问题，算法的实现与数据的存储结构是密切相关的。

#### 1.1.3 数据类型和抽象数据类型

数据类型是一个值的集合和定义在这个值的集合上的一组操作的总称，通常它可看作是高级程序设计语言中已经实现的数据结构。例如，C 语言中的整型变量，其值的集合为某个区间上的整数（区间大小依赖于不同的机器），定义在其上的操作为加、减、乘、除和取模等算术运算。按“值”的不同特性，在高级程序语言中可分为：① 原子类型，其值不可分解，例如，C 语言中的基本类型（整型、实型、字符型和枚举类型）、指针类型和空类型；② 结构类型，其值是由若干个成分按某种结构组成的，故可分解，其成分可以是非结构的，也可以是结构的，例如，数组的值由若干分量组成，每个分量可能是整数，或者是数组。

抽象数据类型（abstract data type，ADT）是指一个数学模型以及定义在该模型上的一组操作。抽象数据类型的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何表示和实现无关，也即，不论其内部结构如何变化，只要它的数学特性不变，都不会影响其外部的使用。抽象数据类型可表示为一个三元组  $(D, R, P)$ ，式中， $D$  是数据对象， $R$  是  $D$  上的关系集， $P$  是对  $D$  的基本操作集。本书采用以下格式定义抽象数据类型：

```
ADT 抽象数据类型名 {
    数据集合：<数据对象的定义>
    数据关系：<数据关系的定义>
    数据操作：<数据操作的定义>
} ADT 抽象数据类型名
```

其中，数据对象和数据关系的定义用伪码描述，基本操作的定义格式为：

```
数据操作名(参数表)
输入：<输入条件描述>
输出：<输出结果描述>
```

**【例 1-2】** 抽象数据类型（ADT）的定义、特点、实现与数据结构的区别是什么？

ADT 是高级程序设计语言中数据类型概念的推广，它是一个数学模型和定义在该模型上操作集合的总称。ADT 的实现方法是，将 ADT 转换成现有高级程序设计语言的说明语句，加上对应于该 ADT 中每个操作的过程或函数，也即，用现有高级程序设计语言能够支持的适当数据结构来表示 ADT 中的数学模型，并用一组过程或函数来实现定义在该模型上的各个操作。数据结构则是利用该语言的基本数据类型和复合数据的构造机制来构成的。例如，数组和记录就是 C++ 语言中两种主要的复合数据的构造机制。根据 ADT 定义，如果在

相同的数学模型上定义两个不同的操作集，则认为它们代表两个不同的抽象数据类型。故相同的数学模型以及在其上所定义的操作有可能在不同的 ADT 中出现。

## 1.2 算法和算法分析

### 1.2.1 算法的描述

算法（algorithm）定义：为了解决某一类问题而设计的一个有限长的操作序列。一个算法必须满足以下 5 个重要特性。

- ① 有穷性。算法对于任意合法的输入值，在执行有限步之后一定能结束。
- ② 确定性。算法中的每一个操作必须有确切的含义，无二义性，并在任何条件下，算法都只有一条执行路径。
- ③ 可行性。算法中的所有操作都可通过已经实现的基本运算有限次地实现。
- ④ 输入。算法具有零个或多个输入，这些输入为一组特定的数据对象集合。
- ⑤ 输出。算法具有一个或多个输出，它是一组与“输入”有确定关系的量值。

### 1.2.2 算法设计的要求

#### （1）正确性（correctness）

算法的执行结果应当满足预先规定的 4 个要求：①程序不含语法错误；② 程序对于几组输入数据能够得出满足规格说明要求的结果；③程序对于精心选择的典型、苛刻且带有刁难性的几组输入数据能够得出满足规格说明要求的结果；④程序对于一切合法的输入数据都能产生满足规格说明要求的结果。

#### （2）可读性（readability）

算法应有助于人们阅读、理解和调试，晦涩难懂的算法易于隐藏较多错误，难以调试和修改。

#### （3）健壮性（robustness）

当输入不合法的数据时，算法能够做出适当的反应或处理，不至于产生莫名其妙的结果。同时，处理出错的方法应该是返回一个表示错误或错误性质的值，并终止程序的执行，以便在更高的抽象层次上进行处理。

#### （4）时空效率（efficiency）

要求算法执行的时间应该尽可能短、算法执行过程中占用的存储空间应该尽可能少。时空要求与求解问题的规模有关，两者通常相互矛盾，因此，应在它们之间有所平衡。

### 1.2.3 算法分析

算法分析的两个主要方面是分析算法的时间复杂度和空间复杂度，主要考察算法的时间效率和空间效率，以便比较和改进算法。通常，在算法的运算空间较为充裕的情况下，更多地关注算法的时间复杂度。

#### 1. 时间复杂度

算法执行的时间可通过依据该算法编制的程序在计算机上从开始运行到结束运行所消

耗的时间来度量，也就是算法中每条语句的执行时间之和。由于每条语句的执行时间是该语句重复执行的次数或频度（frequency count）与该语句执行时间的乘积，而语句的执行时间又与机器性能、编译程序等诸多因素有关，难以统一和确定。因此，假设每条语句的执行时间为一个单位时间，则算法的执行时间为算法中所有语句的频度之和。

一般而言，算法中基本操作的频度是问题规模  $n$ （如算法所处理的矩阵的阶数，线性表的长度）的某个函数  $f(n)$ ，算法的时间量度记作  $T(n)=O(f(n))$ ，它表示随问题规模  $n$  的增长，算法的执行时间增长率与  $f(n)$  的增长率相同，称为算法的渐进时间复杂度（asymptotic time complexity），简称时间复杂度。同时，要全面地分析算法，需要分别考虑算法在最坏情况、最好情况以及平均情况下的时间代价。对于最坏情况下的时间复杂度，主要采用大写数学符号  $O$  表示法来描述。一般定义为：当且仅当存在正整数  $c$  和  $n_0$ ，使得  $T(n) \leq c f(n)$  对所有的  $n \geq n_0$  成立，则称该算法的渐进时间复杂度为  $T(n)=O(f(n))$ 。

在一般情况下，对于一个问题（算法）只需选择一种基本操作来讨论算法的时间复杂度即可，有时也需要同时考虑几种基本操作，甚至可对不同的操作赋以不同权值，以反映执行不同操作所需的相对时间，这种做法便于综合比较解决同一问题的两种完全不同的算法。由于算法的时间复杂度考虑的只是对于问题规模  $n$  的增长率，因此在难以精确计算基本操作执行次数（或语句频度）的情况下，只需求出它关于  $n$  的增长率或阶即可。

**【例 1-3】** 如何进行算法的时间复杂度分析。

首先介绍计算增长率的加法规则和乘法规则。设  $T_1(n)$  和  $T_2(n)$  分别是程序段  $P_1$  和  $P_2$  的运行时间，且  $T_1(n)=O(f(n))$ ， $T_2(n)=O(g(n))$ ，即  $T_1(n)$  是  $f(n)$  的函数， $T_2(n)$  是  $g(n)$  的函数（ $O$  函数定义见后），则执行  $P_1$  之后紧接着执行  $P_2$  的运行时间为： $T_1(n)+T_2(n)=O(\max\{f(n), g(n)\})$ ，称为加法规则； $T_1(n) \times T_2(n)=O[f(n) \times g(n)]$ ，称为乘积规则。一般来说，分析程序的时间复杂度是，先求出各模块（各语句）的运行时间，再求整个程序的运行时间，它可表示成唯一参数——输入数据的规模  $n$  的函数。具体可遵循以下规则。

① 每个赋值或读/写语句的运行时间通常是  $O(1)$ 。如果赋值语句的右部为函数调用，则要考虑计算函数值所消耗的时间。

② 序列语句的运行时间由加法规则确定。

③ 语句 if B then  $S_1$  else  $S_2$  的运行时间为条件 B 的测试时间（通常取  $O(1)$ ）加上两个分支语句  $S_1$ 、 $S_2$  运行时间的较大者。若无 else  $S_2$ ，则只需加上  $S_1$  的运行时间。

④ 循环语句的运行时间是循环体本身的运行时间和计算循环参数、测试循环终止条件及跳回循环开头所花的时间，后一部分通常取  $O(1)$ 。遇到多层循环时，要由内层向外层逐层分析。在分析外层循环时间时，内层循环的运行时间应该是已知的，可把内循环看成是外循环体的一部分。

⑤ 若程序中只包含非递归过程，则从没有调用语句（对函数也认为是调用）的过程开始，计算所有这种过程的运行时间。然后考虑有调用语句的任一过程 P，在 P 调用的全部过程的运行时间都算完之后，即可开始计算 P 的运行时间。若在程序中有递归过程，则可令每个递归过程对应于一个未知的时间开销函数  $T(n)$ ，其中  $n$  是过程参数的长度。之后，列出一个关于  $T$  的递归方程并求解之。

**【例 1-4】** 下面是一个  $n \times n$  阶矩阵  $A$  自乘得到  $B=A \times A$  的算法，分析其时间复杂度。

---

```

int A[n][n]; //全局数组
void mtxmlt()
{
    int B[n][n]; //语句频度
    for (int i=0; i<n; i++) //n+1
        for(int j=0; j<n; j++) //n×(n+1)
            {
                B[i][j]=0; //n×n
                for(int k=0; k<n; k++) //n²×(n+1)
                    B[i][j]=B[i][j]+A[i][k]*A[k][j]; //n×n×n
            }
}

```

---

时间复杂度  $T(n)=n+1+n(n+1)+n\times n+n^2(n+1)+n\times n\times n=2n^3+3n^2+2n+1$ 。当  $n\rightarrow\infty$  时,  $T(n)\propto n^3$ , 故算法时间复杂度的数量级为  $O(n^3)$ 。

**【例 1-5】** 已知算法如下, 求带下划线语句的频度。

---

```

int i=0;
while((i<n)&&(x!=A[i])) i++;
if(A[i]==x) return i;

```

---

在此程序段中, 语句的频度不仅是  $n$  的函数, 而且与  $x$  及数组  $A$  中各分量的值有关。在这种情况下, 通常考虑最坏的情况。由于  $\text{while}$  循环执行的最大数为  $n-1$ , 因此下划线语句频度为  $n-1$ 。

**【例 1-6】** 分析计算  $n!$  的递归函数  $\text{fact}(n)$  的时间复杂度。

---

```

long fact(int n)
{
    if (n==1) return 1;
    return fact(n-1)*n;
}

```

---

递归函数  $\text{fact}(n)$  的输入规模是  $n$ , 设  $T(n)$  是  $\text{fact}(n)$  的时间开销函数。在上述算法中,  $\text{if}$  语句条件测试及语句  $\text{return}$  的运行时间是  $O(1)$ , 递归调用  $\text{fact}(n-1)$  的运行时间是  $T(n-1)$ 。假设两个整数相乘和赋值操作的运算时间是  $O(1)$ , 故  $\text{return fact}(n-1)*n$  的运行时间是  $O(1)+T(n-1)$ 。因此, 对于常数  $C$  和  $D$  有  $T(n)=D, n\leq 1; T(n)=C+T(n-1), n>1$ 。现在来解这个递归方程。设  $n>2$ , 对上式中  $T(n-1)$  进行展开有  $T(n-1)=C+T(n-2)$ , 代入  $T(n)$  中, 有  $T(n)=2C+T(n-2)$ , 再展开  $T(n-2)$  得  $T(n)=3C+T(n-3)$ 。一般有  $T(n)=iC+T(n-i), n>i$ 。最后, 当  $i=n-1$  时, 得  $T(n)=C(n-1)+T(1)=C(n-1)+D$ 。当  $n\rightarrow\infty$  时,  $T(n)\propto n$ 。

## 2. 空间复杂度

算法在执行时需要占用一定的存储空间, 这些空间除了包括程序、输入数据、常数、变量所占的空间外, 还包括算法对输入数据进行运算以及为实现运算所需信息的额外空间。额外空间与算法的质量密切相关, 好的算法既节省时间又节省额外空间。如果算法的输入数据所占用的空间只取决于问题本身, 与算法无关, 则算法的存储空间只需要分析除输入数据和程序之外的额外空间; 否则, 应同时考虑输入数据本身所需空间 (与输入数据的表示形式



有关)。通常,只有完成同一功能的几个算法之间才具有可比性,因此这些输入数据所占用的空间不用进行比较,只需比较那些辅助的或附加的存储空间即可。

类似于算法的时间复杂度,一般以空间复杂度(space complexity)作为算法所需存储空间的量度,记作 $S(n)=O(f(n))$ ,其中 $n$ 为问题的规模(或大小)。在大多数的算法设计中,时间效率和空间效率两者很难兼得,设计者往往需要根据具体的问题进行权衡,有时会用更多的存储空间来换取时间,有的时候又会用增加算法执行时间来减少所需的存储空间。

**【例 1-7】** 如何进行算法的空间复杂度分析。

对算法占用存储空间的分析类似于时间复杂度。估计渐近空间复杂度,称为空间复杂度。由于问题中原始数据所占用的空间与算法无关,故一般考虑空间复杂度时只估算算法中所需增添的辅助空间。例如,例 1-5 中除原始数据外,只用了两个变量 $x$ 和 $i$ 的辅助空间。因为与问题的规模 $n$ 无关,所以空间复杂度 $S(n)=O(1)$ 。

在一般情况下,算法的时间和空间开销是一对矛盾。要想空间比较节约,往往时间消耗就大,反之亦然。具体在一个问题中到底注重哪一方面,这要看实际的需要和可能而定。在本书中,着重考虑时间因素,而假设内存足够大。因为在求解实际问题中,当输入量急剧增加时,如果没有高效率的算法,单纯依靠提高计算机的速度,有时是无法达到要求的。

**注:** $O$ 是数学符号,它的定义是,若 $f(n)$ 是正整数 $n$ 的一个函数,则 $T(n)=O(f(n))$ 表示存在一个正的常数 $M$ ,使得当 $n \geq n_0$ 时都满足 $|T(n)| \leq M|f(n)|$ ,即表明算法所需执行时间是 $f(n)$ 的常数倍。如例 1-6 中,当 $n \geq 1$ 时, $T(n) \leq 3n$ ,即 $n_0=1, M=3, f(n)=n$ 。

## 本章总结

### 1. 学习要点

本章主要介绍了:数据结构和抽象数据类型(ADT)等基本概念及术语定义;算法的描述方法与设计要求;从时间和空间角度,分析算法效率和存储空间需求的方法;算法时间复杂度及空间复杂度的表示等。主要学习要点如下:① 数据、数据元素、数据对象、数据结构等基本概念及术语的确切定义和相互关系;② 数据的逻辑结构与物理结构的基本组织形式和实现方式以及抽象数据类型(ADT)的概念;③ 算法的重要特性;④ 算法设计的基本要求;⑤ 计算算法的语句频度与估算算法时间复杂度(数量级)和空间复杂度(数量级)的方法及表示方法。

### 2. 基本要求

- (1) 掌握数据项、数据元素、原子元素、数据对象、数据结构之间的区别及关系。
- (2) 掌握数据的逻辑结构与数据元素之间的逻辑关系和数据存储结构的含义。
- (3) 理解逻辑结构的 4 种基本组织形式和存储结构的 4 种不同表示方法及其特点。
- (4) 掌握算法分析的方法和时空复杂度的表示。
- (5) 弄清算法的概念、分类、与程序的区别、描述方法和工具(C++语言)。
- (6) 弄清算法与运算、运算的实现、操作的相互关系和区别。
- (7) 掌握算法的时间复杂度和空间复杂度的含义及数量级的概念,计算方法和表示形式。
- (8) 弄清最坏情况下算法时间复杂度和平均时间复杂度的定义、区别、估算方法。

### 3. 重点与难点

重点是：数据结构的概念、逻辑结构和存储结构的组织和表示形式，描述算法的 C++ 语言；难点是：最坏情况下算法的时间复杂度分析。

## 习题 1

1-1 什么是数据、数据元素、原子元素？它们有何区别？

1-2 什么是数据类型、原子数据类型、结构数据类型、抽象数据类型、虚拟数据类型、固有数据类型？它们之间的关系怎样？

1-3 数据结构与软件的关系是什么？解决实际问题时，选取（设计）数据结构的总则是什么？

1-4 如何理解数据（或抽象数据）类型及其实例？C++语言的数据类型 `char` 与 `boolean` 的值域是什么？分别定义了什么操作？

1-5 抽象数据类型（ADT）与面向对象方法有何关系？ADT 的说明如何编写？使用 ADT 的优点有哪些？

1-6 为什么说数据元素之间的逻辑关系是数据内部组织的主要方面？什么是随机存取、顺序存取、直接存取？

1-7 逻辑结构与存储结构是什么关系？有何区别？

1-8 运算与运算实现是什么关系？有哪些相同点和不同点？什么是操作？

1-9 试描述数据结构的概念与高级程序设计语言中数据类型概念的区别。C++语言与 Visual C++语言的区别。

1-10 举一个数据结构的例子，叙述其逻辑结构、存储结构和运算三方面的内容。

1-11 算法与程序有何不同？为何要引出算法的概念？如何评价算法的时间、空间复杂度及算法的好坏？

1-12 分析下列程序段的时间复杂度。

---

```
void odd(int n)
{
    int i, j, x=0, y=0;
    for (i=1; i<=n; i++)
        if odd(i)
            { for(j=i; j<=n; j++) x++;
              for(j=1; j<=i; j++) y++;
            }
}

void recursive(int n)
{ if (n<=1) return 1; else return(recursive(n-1)+recursive(n-1));}
```

---

1-13 设  $n$  为正整数，试确定下列各程序段中带下划线语句的频度。

(1) `int i=1, k=0; while (i<=n-1) { k=k+10*i; i=i+1; }`

(2) `int i=1, j=0; while ((i+j)<=n) if (i>j) j++; else i++;`

(3) `int x=91, y=100;`

```
while (y>0) { if (x>100) { x=x-10; y--; } else x++; }
```

(4) int x=0;

```
for (int i=1; i<=n; i++)
```

```
for (int j=1; j<=i; j++)
```

```
for (int k=1; k<=j; k++) x++;
```

(5) int x=n, y=0; //n>1

```
while (x>=(y+1)*(y+1)) y++;
```

1-14 考虑下列函数:  $f_1(n)=n^2$ ,  $f_2(n)=n^2+1000n$ ,  $f_3(n)=n$  (如果  $n$  是奇数) 或者  $f_3(n)=n^3$  (如果  $n$  是偶数),  $f_4(n)=n$  (如果  $n \leq 100$ ) 或者  $f_4(n)=n^3$  (如果  $n \geq 100$ ), 指出对于不同的  $i$  和  $j$  (比如  $i, j=1, 2, 3, 4$ ), 是否存在  $f_i(n)=O(f_j(n))$ ,  $f_i(n)=\Omega(f_j(n))$ ,  $\Omega$  的定义见习题 1-17。

1-15 设有数据的逻辑结构为  $K=(D, S)$ ,  $D=\{d_1, d_2, \dots, d_9\}$ ,  $S=\{<d_1, d_2>, <d_1, d_8>, <d_2, d_3>, <d_2, d_4>, <d_2, d_5>, <d_3, d_9>, <d_5, d_6>, <d_8, d_9>, <d_9, d_7>, <d_4, d_7>, <d_4, d_6>\}$ 。试画出这个逻辑结构的图示。并确定对应关系集合  $S$  中哪些结点是开始结点, 哪些结点是终端结点?

1-16 考虑以下函数:  $g_1(n)=n^2$  (当  $n$  为偶数时) 或者  $g_1(n)=n^3$  (当  $n$  为奇数时),  $g_2(n)=n$  (当  $0 \leq n \leq 100$  时),  $g_2(n)=n^3$  (当  $n > 100$  时),  $g_3(n)=n^{2.5}$ , 指出  $g_i(n)$  是否为  $O(g_j(n))$ , 以及  $g_j(n)$  是否为  $\Omega(g_i(n))$ ,  $i, j=1, 2, 3$  ( $O$  的定义见例 1-7,  $\Omega$  的定义见习题 1-17)。

1-17 设  $T_1(n)$  是  $\Omega(f(n))$ ,  $T_2(n)$  是  $\Omega(g(n))$ , 问以下论断是否正确:

①  $T_1(n)+T_2(n)$  是  $\Omega\{\max[f(n) \times g(n)]\}$ ;

②  $T_1(n)+T_2(n)$  是  $\Omega\{\max[f(n) \times g(n)]\}$ 。

( $\Omega$  的概念定义为: 若  $f(n)$  是  $\Omega[g(n)]$ , 则表示存在着常数  $n_0$  与  $c>0$ , 使得  $f(n) \geq c \times g(n)$  对于一切  $n \geq n_0$  成立。)

1-18 指出以下算法中的错误和低效(费时)之处, 并将它改写成为一个既正确又高效的算法。

---

```
int last,i,k;int a[n];          //本算法从 a[0]~a[last]中删除第 i 个元素起的 k 个元素
if ((i<0) || (i>last) || (k<0) || (last>n-1))
{   cerr<< "error:argument error" <<endl; exit(1); }
else for (int count=1;count<=k;count++) //删除一个元素
        for (int j=last;j>=i+1;j--) { a[j-1]=a[j];last--;}

```

---

1-19 已知  $k$  阶 Fibonacci 序列定义为  $F_0=0, F_1=0, \dots, F_{k-2}=0, F_{k-1}=1, F_n=F_{n-1}+F_{n-2}+\dots+F_{n-k}, n=k, k+1, \dots$ , 试编写求任意  $k$  阶 Fibonacci 序列的算法(要求序列计算时,  $F_n$  的值不大于  $10^5$ ), 并分析算法时间复杂度。

1-20 设计求解下列问题的 C++ 语言算法, 并分析其最坏情况时间复杂度及其数量级。①在数组  $A[1..n]$  中查找值为  $k$  的元素, 若找到, 则输出其位置  $i$  ( $1 \leq i \leq n$ ), 否则输出 0 作为标志。②找出数组  $A[1..n]$  中元素的最大值和次最大值(本题以数组元素的比较为标准操作)。

1-21 试采取逐步求精的方法用 C++ 语言编写求最大公因子的算法, 并分析算法的时间复杂度。

1-22 试编写一个求多项式  $P_n(x)=a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$  的值  $P_n(x_0)$  的算法, 要求所用的乘法次数最小, 并确定算法中每一个语句的频度及整个算法的时间复杂度。

1-23 要求将整数  $1 \sim 100$  的平方根分成两组, 每组各 50 个数, 使得一组中的各数之和尽可能接近另一组的各数之和。如果只有两分钟的机时可用来解决这个问题, 那么应进行什么运算?

1-24 设下面过程的参数  $n$  取值为 2 的正方幂, 亦即  $n=2, 4, 8, 16, \dots$ 。当过程终止时, 变量 count 的

值作为  $n$  的函数应如何表示?

---

```
int n; int x=2, count=0; while (x<n) { x=x*2; count++;}
```

---

1-25 设  $A$  是一个整数数组, 函数  $\max(i, n)$  可求出  $A$  中从  $i$  到  $i+n-1$  的单元中的最大整数。为简便起见, 可假设  $n$  是 2 的方幂。

---

```
int max(int i, int n)
{
    int m1, m2;
    if (n==1) return A[i];
    else { m1=max(i, n/2); m2=max(i+n/2, n/2);
          if (m1<m2) return m2; else return m1; }
}
```

---

(1) 设  $\max$  在最坏情况下的运行时间函数为  $T(n)$ , 这里的  $n$  是  $\max$  的第二个参数, 表示要在  $n$  个整数中寻找最大的整数。试列出  $T(n)$  满足的递归方程, 也即用若干常数及若干项  $T(j)$  ( $j < n$ ) 代表程序  $\max$  中各语句的运行时间, 并用它们表示整个程序的运行时间  $T(n)$ 。

(2) 在大  $O$  意义下给出  $T(n)$  的一个“精确”上界, 即它也可作为大  $\Omega$  意义下的下界。要求所给出的表达式尽可能简单。

1-26 某单位每个职工都有一张职工登记表。设想在任何组合的已知条件下 (如只知道姓名、知道姓名和单位、知道姓名和性别, 等等), 如何存放这些登记表, 以便能用最快的速度找到某个人的登记表。

1-27 某班本学期开设政治、数学、英语、数据结构和计算机原理 5 门课程。 $n$  个学生平均成绩分优、良、及格和不及格 4 个等级: 90 分以上为优, 80 分至 90 分为良, 60 分至 79 分为及格, 60 分以下为不及格。用 C++ 语言写出统计分析算法, 并分析算法的时间复杂度。

1-28 已知有实现同一功能的两个等法, 其时间复杂度分别为  $O(2^n)$  及  $O(n^{10})$ 。假设计算机可连续运行  $10^7$  秒, 每秒可执行 C++ 基本语句  $10^5$  次。试问: 在此条件下, 可解问题的规模是多大, 即  $n$  的范围约为多少? 用什么算法好, 并说明理由是什么。

1-29 对下列 4 种折半查找程序进行比较, 这 4 个程序哪些是正确的? 哪些效率更高? 假定下列变量, 以及常量  $n > 0$ 。

程序 A:

---

```
void binsearchA(int a[n], int x, int &k)
{
    int i=0, j=n-1;
    while(i<=j)
    { k=(i+j)/2; if(a[k]==x) return; if(x>a[k]) i=k+1; else j=k-1; }
}
```

---

程序 B:

---

```
void binsearchB(int a[n], int x, int &k)
{
    int i=0, j=n-1;
    while (i<j)
    { k=(i+j)/2; if(a[k]==x) return; if(x>a[k]) i=k; else j=k; }
}
```

---

程序 C:

---

```
void binsearchC(int a[n], int x, int &k)
{   int i=0, j=n-1;
    while(i<=j)   {   k=(i+j)/2; if (x<=a[k]) j=k-1;   if (x>=a[k])   i=k+1; }
}
```

---

程序 D:

---

```
void binsearchD(int a[n], int x, int &k)
{   int i=0, j=n-1;
    while (i<j)   {   k=(i+j)/2; if (x<a[k]) j=k;   else i=k+1; }
}
```

---

(提示: 若所查找的元素存在, 则程序必须终止于  $a[k]=x$ ; 若不存在具有值  $x$  的元素, 则程序必须终止于  $a[k] \neq x$ 。)

# 第2章 线 性 表

线性表是实际应用中最基本、最简单、最常用的一种数据结构，例如，26 个英文字母表(A, B, ..., Z)就是一个线性表，表中的元素是单个字母。线性表的基本特点是，数据元素之间具有一种线性关系，即除第一个元素外，集合中的每个元素均有且只有一个直接前驱元素；除最后一个元素外，集合中的每个元素均有且只有一个直接后继元素。

## 2.1 线性表的类型定义

### 2.1.1 基本概念

线性表是由  $n$  ( $n \geq 0$ ) 个类型相同的数据元素组成的有限序列，通常表示为  $L=(a_1, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$ 。其中， $L$  为线性表名称， $a_i$  为组成该线性表的数据元素， $a_{i-1}$  领先于  $a_i$ ， $a_i$  领先于  $a_{i+1}$ ，称  $a_{i-1}$  是  $a_i$  的直接前驱元素， $a_{i+1}$  是  $a_i$  的直接后继元素。当  $i=1, 2, \cdots, n-1$  时， $a_i$  有且仅有一个直接后继；当  $i=2, 3, \cdots, n$  时， $a_i$  有且仅有一个直接前驱。

线性表的长度就是线性表中元素的个数  $n$  ( $n \geq 0$ )。当  $n=0$  时，称为空表。在非空表中的每个数据元素都有一个确定的位置，如  $a_1$  是第一个数据元素， $a_n$  是最后一个数据元素， $a_i$  是第  $i$  个数据元素。称  $i$  为数据元素  $a_i$  在线性表中的位序。

例如，某大学生命科学学院的学生名册可视为一个线性表，如图 2.1 所示，表中每个数据元素由学号、姓名、性别、年龄、院名称、系名称等数据项组成。从图 2.1 可以看出，学生名册线性表是一个相当灵活的数据结构，它的长度可根据需要增长或者缩短，即对线性表的数据元素不仅可访问，还可进行插入、删除等操作。

学号	姓名	性别	年龄	院名称	系名称
09001001	张丰	男	17	生命科学学院	生物技术学系
09001002	刘小红	女	18	生命科学学院	生物技术学系
09001003	李静	女	18	生命科学学院	细胞及遗传学系
09001004	王瑞	男	18	生命科学学院	细胞及遗传学系
⋮	⋮	⋮	⋮	⋮	⋮

图 2.1 某大学生命科学学院学生名册

### 2.1.2 抽象数据类型描述

抽象数据类型描述见 ADT2-1。

#### ADT2-1 线性表的抽象数据类型描述

```
ADTLinear List {
    数据集合:  $D=\{a_i|a_i/\text{ElemSet}, i=1, 2, \cdots, n, n \geq 0\}$ 
    数据关系:  $R=\{\langle a_{i-1}, a_i \rangle|a_{i-1}, a_i/D, i=2, \cdots, n\}$ 
    数据操作:
```

Init\_List(&L) //初始化线性表  
 输入：空。输出：构造一个空的线性表 L。

Destroy\_List(&L) //撤销线性表  
 输入：线性表 L。输出：撤销线性表 L。

Clear\_List(&L) //清空线性表  
 输入：线性表 L。输出：线性表 L 重置为空表。

List\_Empty(L) //判断线性表是否为空  
 输入：线性表 L。输出：若线性表 L 为空表，则返回 TRUE，否则返回 FALSE。

List\_Length(L) //求线性表的长度  
 输入：线性表 L。输出：线性表 L 中数据元素个数。

Get\_Elem(L, i, &e) //取线性表中第 i 个数据元素的值  
 输入：线性表 L,  $1 \leq i \leq \text{ListLength}(L)$ 。  
 输出：用 e 返回线性表 L 中第 i 个数据元素的值。

LocateElem(L, e, compare()) //返回给定值在线性表中的位置  
 输入：线性表 L, compare() 是数据元素相等判定函数。  
 输出：线性表 L 中第 1 个与 e 满足相等关系的数据元素的位序。若这样的数据元素不存在，则返回值为 0。

Prev\_Elem(L, cur\_e, &pre\_e) //返回当前元素的前一个元素值  
 输入：线性表 L。  
 输出：若 cur\_e 是线性表 L 的数据元素，且不是第一个，则用 pre\_e 返回它的直接前驱元素；否则操作失败，pre\_e 无定义。

Next\_Elem(L, cur\_e, &next\_e) //返回当前元素的后一个元素值  
 输入：线性表 L。  
 输出：若 cur\_e 是线性表 L 的数据元素，且不是最后一个，则用 next\_e 返回它的直接后继元素；否则操作失败，next\_e 无定义。

Insert\_List(&L, i, e) //在线性表的第 i 个位置之前插入数据元素  
 输入：线性表 L,  $1 \leq i \leq \text{ListLength}(L)+1$ 。  
 输出：在线性表 L 中第 i 个位置之前插入新的数据元素 e，线性表 L 的长度加 1。

Delete\_List(&L, i, &e) //删除线性表中第 i 个数据元素  
 输入：线性表 L 非空,  $1 \leq i \leq \text{ListLength}(L)$ 。  
 输出：删除 L 中第 i 个数据元素，并用 e 返回其值，线性表 L 的长度减 1。

Traverse\_List(L, visit()) //遍历线性表  
 输入：线性表 L。  
 输出：依次对线性表 L 的每个数据元素调用 visit() 进行访问。一旦 visit() 失败，则操作失败。

}ADT LinearList

### 2.1.3 线性表抽象类

将线性表视为一个抽象对象/类（亦称接口），即不考虑它的具体数据结构存储，不考虑基本操作的实现，只考虑它的基本操作的接口（输入/输出）。程序 2-1 给出的线性表抽象类

描述是基本的操作，读者可自行扩充。

程序 2-1 线性表抽象类 LinearList

```
#include<iostream.h>
template<class T>
class LinearList
{
public:
    LinearList(); //构造函数
    virtual ~LinearList()=0; //析构函数
    virtual bool IsEmpty() const { return length==0; }
    virtual int Length() const { return length; }
    virtual bool Find(int k, T &x) const=0; //返回第 k 个元素至 x 中
    virtual bool Delete(int k, T &x)=0; //删除第 k 个元素并将它返回至 x 中
    virtual bool Insert(int k, const T &x)=0; //在第 k 个元素之后插入 x
    virtual void Output(ostream &out) const=0;
protected:
    long length;
};
```

## 2.1.4 异常类NoMem和OutOfBounds

如果分配内存失败，应引发一个异常。有时异常可能由 **new** 引发，而有时则需要编程者来引发。为了在所有情形下都能引发一个异常，本节定义一个异常类 **NoMem**，非法操作将会简单地引发一个类型为 **NoMem** 的异常。另外，在删除操作中，为了从一个表中删除第  $k$  个元素，需要先确认表中包含第  $k$  个元素，然后再删除这个元素。如果表中不存在第  $k$  个元素，则应出现一个越界异常，定义为 **OutOfBounds**，每当正在执行的函数中任何一个参数超出所期望的范围时，就引发这种类型的异常。这两个异常的定义见程序 2-2。

程序 2-2 内存不足异常 NoMem 与越界异常 OutOfBounds

```
//内存不足
#include<iostream.h>
class NoMem{
public:
    NoMem() { cout<<"memory is not enough"; };
};
//越界
class OutOfBounds
{
public:
    OutOfBounds() { cout<<"out of bounds"<<endl; };
};
```



## 2.2 线性表的顺序存储结构

### 2.2.1 基本概念

#### 1. 线性表的顺序表示

线性表的顺序表示，是指用一组地址连续的存储单元依次存储线性表中的数据元素。设  $a_1$  的存储地址为  $\text{Loc}(a_1)$ ，每个数据元素占用  $d$  个字节存储单元，则第  $i$  个数据元素的地址为  $\text{Loc}(a_i)=\text{Loc}(a_1)+(i-1)\times d$ ， $1\leq i\leq n$ ，如图 2.2 所示。

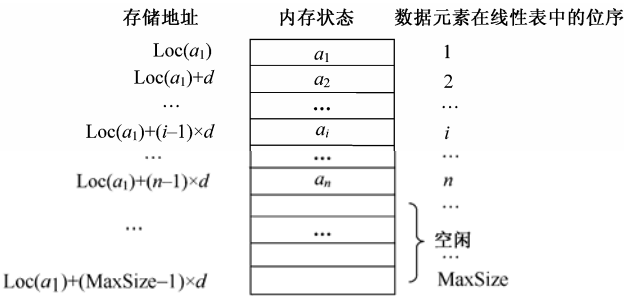


图 2.2 线性表的顺序存储结构示意图

$\text{Loc}(a_1)$ 通常称作线性表的起始位置或基地址。只要确定了存储线性表的起始位置，线性表中任一个数据元素都可随机存取。因此，线性表的顺序存储结构是一种随机存取的存储结构。线性表的这种机内表示称为线性表的顺序存储结构或顺序映射。通常，称这种存储结构的线性表为顺序表，其特点是以数据元素在计算机内“物理位置相邻”来表示线性表中数据元素之间的逻辑关系。

#### 2. 线性表的动态分配顺序存储结构的描述

线性表的长度可变，而且最大存储空间随问题的不同而不同，因此需要动态地分配线性表的空间。在 C++语言中，可用动态分配的一维数组来表示，描述见程序 2-3。

程序 2-3 动态分配顺序存储结构的线性表类

```
#include "LinearList.h"
template<class T>
//上面是模板声明，表明 T 是一个可变(调)类型。具体的类型在使用 LinearListSqu 时动态决定
//有了这个声明，LinearListSqu 中可直接将 T 作为已知类型使用
class LinearListSqu: public LinearList<T> //表示从 LinearList<T>派生
{
public:
    LinearListSqu(int MaxListSize=10);    //构造函数
    virtual ~LinearListSqu() { if (element) delete[] element; } //析构函数
    virtual bool IsEmpty() const{ return length==0; }
    virtual int Length() const{ return length; }
```

```

virtual bool Find(int k, T &x) const;    //返回第 k 个元素至 x 中
virtual bool Delete(int k, T &x);      //删除第 k 个元素并将它返回至 x 中
virtual bool Insert(int k, const T &x); //在第 k 个元素之后插入 x
virtual void Output(ostream &out) const;
protected:
    long length;                        //线性表当前长度
    long MaxSize;
    T*element;                          //一维动态数组，其元素类型为可变类型 T
};

```

---

## 2.2.2 基本操作

### 1. 线性表的初始化、查找和输出

描述见程序 2-4。

程序 2-4 线性表的初始化和查找

```

template<class T>
LinearListSqu<T>:: LinearListSqu (int MaxListSize)
{    //线性表的构造函数，初始化线性表
    MaxSize=MaxListSize;
    try{ element=new T[MaxSize]; }
    catch(...) { throw NoMem( ); }
    length=0;
}

template<class T>
bool LinearListSqu<T>:: Find(int k, T &x) const
{    //把第 k 个元素取至 x 中
    //如果不存在第 k 个元素，则返回 false；否则返回 true
    if (k<1||k>length) return false; //不存在第 k 个元素
    x=element[k-1];
    return true;
}

template<class T>
void LinearListSqu<T>:: Output(ostream& out) const
{ for (int i=0; i<length; i++) out<<element[i]<<" "; } //把表输送至输出流
//重载<<
template<class T>
ostream& operator<<(ostream& out, const LinearListSqu<T>& x)
{    x.Output(out);
    return out;
}

```

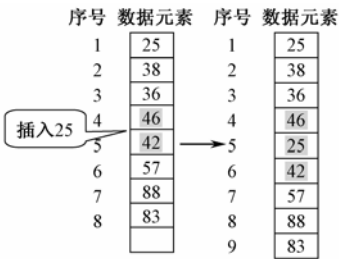
---

### 2. 插入和删除操作

插入和删除是线性表的基本操作，下面分别进行详细讨论。

(1) 插入操作。线性表的插入操作是指在线性表的第  $i-1$  个数据元素和第  $i$  个数据元素之间插入一个新的数据元素  $b$ ，其结果是使长度为  $n$  的线性表  $(a_1, \cdots, a_{i-1}, a_i, \cdots, a_n)$  变成长度为  $n+1$  的线性表  $(a_1, \cdots, a_{i-1}, b, a_i, \cdots, a_n)$ ，并且数据元素  $a_{i-1}$  和  $a_i$  之间的逻辑关系发生了变化。在线性表的顺序存储结构中，由于逻辑上相邻的数据元素在物理位置上也相邻，因此需将第  $i \sim n$  (共  $n-i+1$ ) 个元素向后移动一个位置，才能反映这个逻辑关系的变化。如图 2.3 所示，为了在线性表的第 4 个和第 5 个元素之间插入值为 25 的数据元素，则需要将第 5~8 个元素依次往后移动一个位置。

为了在表中第  $k$  个元素之后插入一个新元素，首先需要把第  $k+1 \sim \text{length}$  个元素向后移动一个位置，然后把新元素插入到  $k+1$  位置。程序 2-5 中给出了插入操作的 C++ 代码。



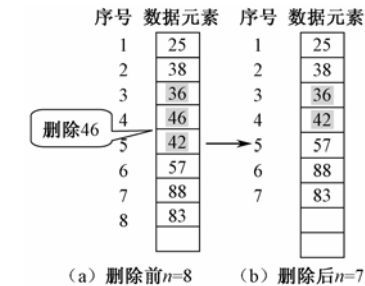
(a) 插入前  $n=8$  (b) 插入后  $n=9$

图 2.3 线性表插入前后的情况

程序 2-5 插入算法

```
template<class T>
bool LinearListSqu<T>:: Insert(int k, const T& x)
{
    //在第 k 个元素之后插入 x
    //如果不存在第 k 个元素，则引发异常 OutOfBounds
    //如果表已经满，则引发异常 NoMem
    if (k<0||k>length) throw OutOfBounds( );
    if (length==MaxSize) throw NoMem( );
    //向后移动一个位置
    for (int i=length-1; i>=k; i--) element[i+1]=element[i];
    element[k]=x; length++;
    return true;
}
```

注意，在插入操作期间，可能出现两类异常：第一类异常发生的情形是没有正确指定插入点，如在插入新元素之前，表中元素个数少于  $k-1$  个，或者  $k<0$ ，在这种情形下，引发一个 `OutOfBounds` 异常；当表已经满时，会发生第二类异常，此时数组中没有剩余的空间来容纳新元素，因此将引发一个 `NoMem` 异常。`Insert` 操作的时间复杂度为  $O(\text{length}-k)$ 。



(a) 删除前  $n=8$  (b) 删除后  $n=7$

图 2.4 线性表删除前后的情况

(2) 删除操作。与线性表的插入运算相反，线性表的删除操作是使长度为  $n$  的线性表  $(a_1, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$  变为长度为  $n-1$  的线性表  $(a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n)$ ，并且数据元素  $a_{i-1}$ 、 $a_i$  和  $a_{i+1}$  之间的逻辑关系也会发生变化，需要把第  $i+1 \sim n$  个元素 (共  $n-i$  个元素) 依次向前移动一个位置来反映这个变化。如图 2.4 所示，为了删除第 4 个元素，需要将第 5~8 个元素依次向前移动一个位置。程序 2-6 中给出了删除操作的 C++ 代码。

```
template<class T>
bool LinearListSqu<T>:: Delete(int k, T& x)
{    //把第 k 个元素放入 x 中，然后删除第 k 个元素
    //如果不存在第 k 个元素，则引发异常 OutOfBounds
    if (Find(k, x))
    {    //把元素 k+1, ...向前移动一个位置
        for (int i=k; i<length; i++)  element[i-1]=element[i]; length--;
        return true;
    }
    else throw OutOfBounds( );
}
```

(3) 时间复杂度分析。从上述算法可见，当在顺序存储结构的线性表中某个位置上插入或删除一个数据元素时，其时间主要耗费在移动元素上，而移动元素的个数取决于插入或删除元素的位置。假设  $p_i$  是在第  $i$  个元素之前插入一个元素的概率，则在长度为  $n$  的线性表中插入一个元素时所需移动元素次数的期望值  $E_{\text{insert}} = \sum_{i=1}^{n+1} p_i (n-i+1)$ 。

不失一般性，若在线性表的任何位置插入元素都是等概率的，即  $p_i=1/(n+1)$ ，上式可化简为  $E_{\text{insert}} = \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{n}{2}$ 。

对于删除过程，假设  $q_i$  是删除第  $i$  个元素的概率，则在长度为  $n$  的线性表中删除一个元素时所需移动元素次数的期望值  $E_{\text{del}} = \sum_{i=1}^n q_i (n-i)$ 。同样假设是等概率的情况，即  $q_i=1/(n+1)$ ，则有  $E_{\text{del}} = \frac{1}{n+1} \sum_{i=1}^n (n-i) = \frac{n-1}{2}$ 。由此可见，在顺序存储结构的线性表中插入或删除一个数据元素，平均约需移动表中一半元素。若表长为  $n$ ，则上面的插入算法和删除算法的时间复杂度均为  $O(n)$ 。

程序 2-7 是一个使用类 LinearList 的 C++ 程序，它假定程序 2-3 至程序 2-6 均存储在 LinearList.h 之中，且异常类定义位于文件 exception.h 之中。该示例完成以下操作：创建一个大小为 5 的整数线性表 L；输出该表的长度（为 0）；在第 0 个元素之后插入 2；在第一个元素之后插入 6 和 8（至此，线性表为 2，6，8）；寻找并输出第一个元素（为 2）；输出当前表的长度（为 3）；删除并输出第一个元素。图 2.5 给出了由程序 2-7 产生的输出。

程序 2-7 采用类 LinearList 的例子

```
#include<iostream.h>
#include "LinearListSqu.h"
#include "LinearList.h"
void main( )
{    try{
        LinearListSqu<int>L(5);
        cout<<"Length="<<L.Length( )<<endl;
```

```

        cout<<"IsEmpty="<<L.IsEmpty( )<<endl;
        bool ok=false; ok=L.Insert(0,2);
        if (!ok) cout<<"ok"<<endl;
        L.Insert(1,6); L.Insert(2,8);
        cout<<"List is "<<L<<endl; cout<<"IsEmpty="<<L.IsEmpty( )<<endl;
        int z;
        L.Find(1,z);
        cout<<"First element is "<<z<<endl; cout<<"Length="<<L.Length( )<<endl;
        L.Delete(1,z);
        cout<<"Deleted element is "<<z<<endl; cout<<"List is "<<L<<endl;
    }

    catch(...) { cerr<<"An exception has occurred"<<endl; }
}

```

```

Length=0
IsEmpty=1
List is 2 6 8
IsEmpty=0
First element is 2
Length=3
Deleted element is 2
List is 6 8

```

图 2.5 程序 2-7 所产生的输出

考虑一些特殊的情形，如需要维持 3 个线性表，而任何时候这三个线性表所拥有的元素总数都不会超过 5000 个，则很有可能在某个时刻，线性表 1 需要 5000 个元素，而在另一时刻，线性表 2 或线性表 3 需要 5000 个元素。若采用类 `LinearList`，则这 3 个线性表中的每一个线性表都需要有 5000 个元素的容量，但即使在任何时刻，都不会使用 5000 以上的元素。

对于这种情况，比较好的方法是把所有的线性表都放在一个数组 `list` 中描述，并使用两个附加的数组 `first` 和 `last` 对这个数组进行索引。图 2.6 给出了在一个数组 `list` 中描述的三个线性表。下面采用更通用的描述方法：假设有  $m$  个线性表，则每个线性表从 1 到  $m$  进行编号，且 `first[i]` 为第  $i$  个线性表中的第一个元素，`last[i]` 是线性表  $i$  的最后一个元素。注意，当第  $i$  个线性表不为空时，有 `last[i]>first[i]`；而当第  $i$  个线性表为空时，有 `last[i]=first[i]`。因此，在图 2.6 所示的例子中，线性表 2 是空表。在数组中，各线性表从左至右按线性表的编号次序 1, 2, 3, ...,  $m$  进行排列。同时，为了避免第一个线性表和最后一个线性表的处理方法与其他线性表不同，定义两个边界表：线性表 0 和线性表  $m+1$ ，其中 `first[0]=last[0]=-1`，`first[m+1]=last[m+1]=MaxSize-1`。

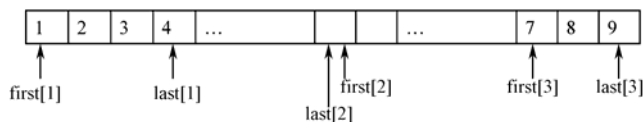


图 2.6 一个数组中的所有线性表

为了在第  $i$  个线性表的第  $k$  个元素之后插入一个元素，首先需要为新元素创建空间。如果 `last[i]=first[i+1]`，则在第  $i$  个线性表和第  $i+1$  个线性表之间没有空间，因此不能把第  $k+1$

个元素至最后一个元素向后移动一个位置。在这种情况下，通过检查关系式  $\text{last}[i-1] < \text{first}[i]$  是否成立，可确定是否有可能把第  $i$  个线性表的  $1 \sim k-1$  元素向前移一个位置；如果这个关系式不成立，要么把线性表  $1 \sim$  线性表  $i-1$  的元素向前移一个位置，要么把线性表  $i+1 \sim$  线性表  $m$  向后移一个位置，然后为线性表  $i$  创建需要增长的空间。当线性表中所有的元素总数小于  $\text{MaxSize}$  时，这种移位的方法是可行的。尽管在一个数组中，描述几个线性表比每个线性表用一个数组来描述的空间利用率更高，但在最坏的情况下，插入操作将耗费更多的时间。事实上，一次插入操作可能需要移动  $\text{MaxSize}-1$  个元素。

**【例 2-1】** 试利用数组来实现线性表的插入、删除、定位操作。

---

```
int insert(int x, int p, int &last, int *list, int maxlength)
//将元素 x 插入到表 list 的位置 p
//last 为表中最后一个元素的下标，maxlength 为表的最大长度
{   int q;
    if (last>=maxlength-1) return 1; //错误：数组已满
    if ((p<0) || (p>last)) return 2; //错误：不合理的位置
    for (q=last; q>p; q--) list[q]=list[q-1];
    list[p]=x; last++;
}

int delete(int p, int *list, int &last) //Delete 从表中删除位置 p 处的元素
{   int q;
    if ((p>last) || (p<0)) return 2; //错误：不合理的位置
    last--;
    for (q=p; q<=last; q++) //将位于 p+1, p+2, ... 的元素向前移 1 位
        list [q]=list[q+1];
}

int locate(int x, int *list, int last) //locate 的返回值为元素 x 在表 list 中的位置
{   int q;
    for (q=0; q<=last; q++) if (list[q]==x) return q;
    return -1; //如果 list 中没有 x
}
```

---

**【例 2-2】** 已知一个线性表中的元素按元素值非递减有序排列，试设计算法删除表中多余的值相同的元素。

算法 1 如下：

---

```
void purgel(int *list, int &last) //last 为表中最后一个元素的下标
{   int i=0; // i 是扫描指示器，赋初值 0
    while (i<=last-1)
    {   if (list[i]!=A[i+1]) i++; //继续往下扫描
        else { for (int j=i+2; j<=last; j++)
                list[j-1]=list[j]; //删除表中第 i+1 个元素
        }
    }
}
```

---

```

        last--;
    }
}
} //该算法的时间复杂度为  $O(n^2)$ 

```

算法 2 如下：

```

void purge2(int *list, int &last)
{
    int j=0, i=1;    //从第二个元素起扫描
    while (i<=last)
    {
        if (list[i]!=list[j]) { list[j+1]=list[i]; j++; }
        //将 list[j]移至正确位置上
        i++;    //继续扫描
    }
}
} //算法 2 时间复杂度为  $O(n)$ ，由此可见，算法 1 的时间效率较算法 2 的低

```

## 2.3 线性表的链式存储结构

线性表的顺序存储结构的特点是，用物理上的相邻关系表达出逻辑上的前驱和后继关系，因此可通过简单的公式获取表中任一元素的地址。但是，在进行插入和删除操作时，需移动大量元素，同时对存储空间的利用也不能很充分。下面介绍线性表的链式存储结构，它不要求逻辑上相邻的元素在物理位置上也相邻，而是通过“链”建立起数据元素之间的次序关系，因此，它没有顺序存储结构所具有的弱点，但同时也失去了顺序表可随机存取的优点。

### 2.3.1 线性链表

#### 1. 线性链表的定义

线性链表是指用一组任意的存储单元存储线性表的数据元素（这组存储单元可以是连续的，也可以是不连续的），通过存储数据元素信息的数据域和直接后继存储位置的指针域构成数据元素  $a_i$  的存储映射，即结点（node），如图 2.7 所示。 $n$  个结点  $a_i$  ( $1 \leq i \leq n$ ) 链接成一个链表，即为线性表( $a_1, a_2, \dots, a_n$ )的链式存储结构。由于此链表的每个结点中只包含一个指针域，故又称线性链表或单链表。

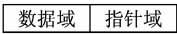


图 2.7 单链表结点结构

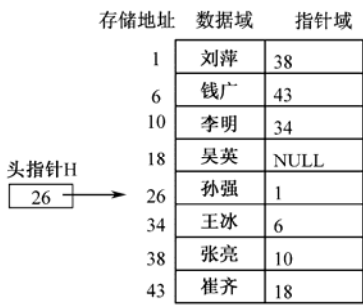


图 2.8 链式存储结构示例

例如，图 2.8 所示为一个科研团队成员的姓名线性表（孙强，刘萍，张亮，李明，王冰，钱广，崔齐，吴英）的线性链表存储结构，整个链表的存取由头指针 H 开始进行，它指向链表中第一个结点的存储位置，线性链表中最后一个结点的指针为“空”（NULL）。

#### 2. 线性表的单链表存储结构的类定义

程序 2-8 中定义了 LinkNode 和 LinearListLink 类，链表对象是线性表抽象类 LinearList（见程序 2-1）的派生

类，代表整个线性链表，它需要记录首结点的地址和链表中当前结点个数等有关链表的信息，并针对其设置有关操作。**LinearListLink<T>**定义为 **LinkNode<T>**的一个友类，所以 **LinearListLink<T>**可访问 **LinkNode<T>**的所有成员（包括是私有成员）。公共成员函数 **Length**、**Find**、**Delete** 和 **Insert** 由抽象类 **LinearList** 继承而来。

#### 程序 2-8 链表的类定义

```
#include "LinearList.h"
template<class T>
class LinearListLink;
//在 LinkNode 类中作为友元，须先声明，此时没有分配类对象空间
template<class T>
class LinkNode
{
public:
    friend class LinearListLink<T>;
private:
    T data; LinkNode<T> *link;
};
template<class T>
class LinearListLink: public LinearList<T> //表示从 LinearList<T>派生
{
public:
    LinearListLink() { first=0; }
    virtual ~LinearListLink();
    virtual bool IsEmpty() const { return first==0; }
    virtual int Length() const;
    virtual bool Find(int k, T&x) const;
    virtual int Search(const T&x) const;
    virtual LinkNode<T> *GetNode(long i); //返回第 i 个结点
    virtual bool Delete(int k, T&x);
    virtual bool Insert(int k, const T&x);
    virtual void Output(ostream& out) const;
    virtual bool Get(LinkNode<T> *node, T&x); //获取结点值
    virtual bool Set(LinkNode<T> *node, const T&x); //设置结点值
    virtual bool GetHead() {return first}; //获得头结点指针
protected:
    LinkNode<T> *first; //指向第一个结点的指针
};
```

假设 **L** 是 **LinearListLink** 型变量，则 **L** 可作为一个单链表的头指针，它指向链表中的第一个结点。若 **L** 为空 (**L=NULL**)，则表示线性表为“空”，线性表的长度为 0。有时，在单链表的第一个结点之前附设一个结点，称为头结点。头结点的数据域可不存储任何信息，也可存储线性表长度等附加信息；头结点的指针域存储指向第一个结点的指针（即第一个元素



结点的存储位置)。

### 3. 线性表的带头结点的单链表存储结构示意图

图 2.9 所示为单链表的一般表示法。

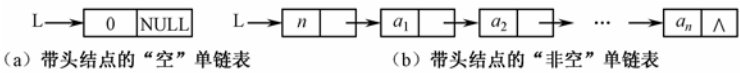


图 2.9 单链表的一般图示表示法

### 4. 基本操作

可采用如下的描述来创建一个空的整数型线性表 `LinearList Link<int>L`。注意，线性表的链表描述不需要指定表的最大长度。

程序 2-9 给出了析构函数的代码，它的时间复杂度为  $O(n)$ ，其中  $n$  为链表的长度。程序 2-10 和程序 2-11 中的代码分别实现了 `Length` 操作和 `Find` 操作。`Length` 的时间复杂度为  $O(n)$ ，`Find` 的时间复杂度为  $O(k)$ 。`Output`（见程序 2-12）的时间复杂度为  $O(n)$ ，它要求对于类型 `T` 必须定义 `<<` 操作。

程序 2-9 删除链表中的所有结点

```
template<class T>
LinearListLink<T>::~~LinearListLink ( )
{
    //链表的析构函数，用于删除链表中的所有结点
    LinkNode<T> *next; //下一个结点
    while (first) { next=first->link; delete first; first=next; }
}
```

程序 2-10 确定链表的长度

```
template<class T>
int LinearListLink<T>:: Length( ) const
{
    //返回链表中的元素总数
    LinkNode<T> *current=first;
    int len=0;
    while (current) { len++; current=current->link; }
    return len;
}
```

程序 2-11 搜索并获得表元素算法

```
template<class T>
bool LinearListLink<T>:: Find(int k, T& x) const
{
    //寻找链表中的第 k 个元素，并将其传送到 x
    //如果不存在第 k 个元素，则返回 false；否则返回 true
    if (k<1) return false;
    LinkNode<T> *current=first;
    int index=1; //current 的索引
```

```

while (index<k && current) { current=current->link; index++; }
if (current) { x=current->data; return true; }
return false; //不存在第 k 个元素
}

```

## 程序 2-12 输出表结点

```

template<class T>
void LinearListLink<T>::Output(ostream& out) const
{ //将链表元素送至输出流
    LinkNode<T> *current;
    for (current=first; current; current=current->link) out<<current->data<<" ";
}
//重载<<
template<class T> ostream& operator<<(ostream& out, const LinearListLink<T> &x)
{ x.Output(out); return out; }

```

## 5. 单链表的插入和删除运算

假设要在线性表的两个数据元素 X 和 Y 之间插入一个数据元素 Z，已知 p 为单链表存储结构中指向结点的指针，如图 2.10 (a) 所示。为插入数据元素 Z，首先要生成一个数据域为 Z 的结点，然后插入到单链表中。根据单链表的逻辑定义，还需修改结点 X 中的指针域，令其指向结点 Z，而结点 Z 中的指针域应指向结点 Y。插入后的单链表如图 2.10 (b) 所示。

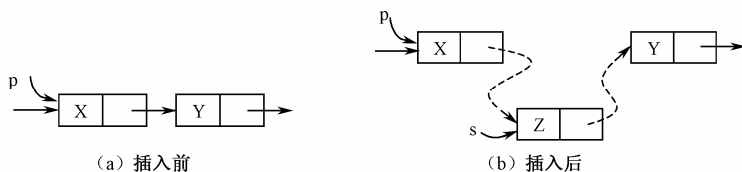


图 2.10 在单链表中插入结点时的指针变化情况

假设 s 为指向结点 Z 的指针，则插入结点操作可以用语句描述如下：s->next=p->next; p->next=s; 反之，如图 2.11 所示在线性表中删除元素 Y 时，仅需修改结点 X 中的指针域来实现链表中逻辑关系的变化即可。

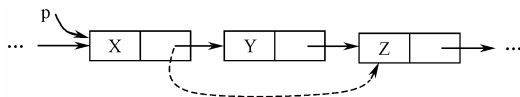


图 2.11 在单链表中删除结点时的指针变化情况

假设 p 为指向结点的指针，则删除结点操作可用语句描述如下：p->next=p->next->next; 可见，在单链表中插入或删除一个结点时，仅需修改指针而无须移动元素。程序 2-13 和程序 2-14 分别为成员函数 Insert 和 Delete 在单链表中的算法实现。

```
template<class T>
bool LinearListLink<T>:: Insert(int k, const T& x)
{    //在第 k 个元素之后插入 x
    //如果不存在第 k 个元素，则引发异常 OutOfBounds
    //如果没有足够的空间，则传递异常 NoMem
    if (k<0) throw OutOfBounds( );
    LinkNode<T> *p=first;  //p 最终将指向第 k 个结点
    //将 p 移至第 k 个元素
    for (int index=1; index<k&&p; index++) p=p->link;
    if (k>0 &&!p) throw OutOfBounds( ); //不存在第 k 个元素
    LinkNode<T> *y=new LinkNode<T>; y->data=x; //插入
    if (k) { y->link=p->link; p->link=y; } //在 p 之后插入
    else { y->link=first; first=y; } //作为第一个元素插入
    if (!y->link) last=y;
    return true;
}
```

#### 程序 2-14 链表删除元素

```
template<class T>
bool LinearListLink<T>:: Delete(int k, T& x)
{    //把第 k 个元素取至 x，然后从链表中删除第 k 个元素
    //如果不存在第 k 个元素，则引发异常 OutOfBounds
    if (k<1||!first) throw OutOfBounds( ); //不存在第 k 个元素
    //p 最终将指向第 k 个结点
    LinkNode<T> *p=first;
    //将 p 移至第 k 个元素，并从链表中删除该元素
    if (k==1) first=first->link; //p 已经指向第 k 个元素，删除之
    else {    LinkNode<T> *q=first; //用 q 指向第 k-1 个元素
        for (int index=1; index<k-1 && q; index++) q=q->link;
        if (!q||!q->link) throw OutOfBounds( ); //不存在第 k 个元素
        p=q->link; //存在第 k 个元素
        if (p==last) last=q; //补充 last
        q->link=p->link;
    } //从链表中删除该元素
    x=p->data; delete p; return true; //保存第 k 个元素并释放结点 p
}
```

容易看出，程序 2-13 和程序 2-14 的时间复杂度均为  $O(n)$ 。因为在第  $i$  个结点之前插入一个新结点或删除第  $i$  个结点，都需要先找到第  $i-1$  个结点，即修改指针的结点，所以它的时间复杂度为  $O(n)$ 。

**【例 2-3】** 定义和建立单链表，并在链表上实现基本的建空表、查找、定位、求表长、插入、删除、置空运算。

## 结点结构定义:

```
struct node { int data;
node *next;
};
```

## 建表算法:

```
void creat_lklist2(node *head)
//直接实现的建表算法, p 是一个 node*类型的变量, 用来指示链入位置
{
    node *p=head; //尾指针置初值
    int x;
    cin>>x; //写入第一个元素
    while (x!='\n') //当前输入不是结束符, 继续链入
    {
        node *q;
        q=new node; q->data=x; //生成一个新结点
        p->next=q; p=q; //新结点链入, 修改尾指针指向新的表尾
        cin>>x; //写入下一个元素
    }
    p->next=NULL; //置尾结点标志
}

node *initiate_lklist( ) //建立一个空表
{
    node *head;
    head=new node; head->next=NULL;
    return head;
}

long length_lklist(node *head) //求表 head 的长度
{
    node *p;
    p=head; long j=0; //p 指向头结点, 计数器计初值
    while (p->next!=NULL)
    {
        p=p->next; j++; } //当未数到尾结点时, 继续“点数”
    return j; //回传表长
}

node *find_lklist(node *head, long i)
//在单链表 head 中查找第 i 个结点。若找到, 则回传指向该结点的指针; 否则回传 NULL
{
    node *p=head;
    long j=0; //置初值
    //未“数”到表尾且未“数”到第 i 个结点时, 继续“点数”
    while ((p->next!=NULL)&&(j<i)) { p=p->next; j++; }
    if (i==j) return p; else return NULL;
}
```

```

long locate_lklist(node *head, int x)
//求表 head 中第一个值等于 x 的结点的序号；不存在这种结点时，结果为 0
{
    node *p=head;
    long j=0; //置初值
    //未到尾结点又未找到值等于 x 的结点时，继续扫描
    while ((p->next!=NULL)&&(p->data!=x)) { p=p->next; j++; }
    if (p->data==x) return j; else return 0;
}

void delete_lklist(node *head, long i) //删除表 head 的第 i 个结点
{
    node *p=find_lklist(head, i-1); //先找待删结点的直接前驱
    if ((p!=NULL)&&(p->next!=NULL))
    {
        node *q=p->next; p->next=q->next; //q 指待删结点，摘除待删结点
        delete q; //释放待删除结点 q
    }
    else cerr<<"不存在第 i 个结点"<<endl;
}

```

---

### 插入算法:

---

```

void insert_lklist(node *head, int x, long i)
//在表 head 的第 i 个位置处，插入一个以 x 为值的新结点
{
    node *p=find_lklist(head, i-1); //先找第 i-1 个结点
    if (p==NULL) cerr<<"不存在第 i 个位置"<<endl; //若第 i-1 个结点不存在，
    //给出出错信息。否则，生成新结点
    else { node *s=new node; s->data=x; s->next=p->next; p->next=s; }
}

```

---

**【例 2-4】** 试设计算法逆置一个用带头结点的链表表示的线性表。  
算法描述如下:

---

```

void invert_linked(node *head)
//head 为带头结点的单链表的头指针，本算法逆置该链表表示的线性表
//逆置后的链表仍带头结点，且头指针不变
{
    node *p=head->next; //p 指向原表中的尚未逆置的第一个结点
    head->next=NULL; //设逆置后的链表为空表
    node *s;
    while (p!=NULL)
    {
        s=p; p=p->next; //s 指向当前逆转的结点
        s->next=head->next; head->next=s; } //插入逆表
}

```

---

## 6. 静态链表

有时可采用一维数组来描述线性链表，其类型说明的 C++ 语言描述见程序 2-15。

```
#include "LinearList.h"
#define MAXSIZE 100 //链表的最大长度
template<class T>
class StaticLinearListLink;
template<class T>
class StaticLinkNode {
    friend class StaticLinearListLink<T>;
private:
    T data; int cur;
};
template<class T>
class StaticLinearListLink: public LinearList<T> //表示从 LinearList<T>派生
{
public: //与前面类定义相似，可自行扩充
    StaticLinearListLink();
    virtual ~StaticLinearListLink();
private:
    StaticLinkNode SLinkList[MAXSIZE];
};
```

在如上描述的链表中，数组的一个分量表示一个结点，同时用游标（指示器 **cur**）代替指针指示结点在数组中的相对位置，如图 2.12 所示。数组的第 0 个分量可看成头结点，其指针域指示链表的第一个结点。图 2.12（b）展示了图 2.12（a）所示线性表在插入数据元素“孙晋”之后的状况，2.12（c）则展示了图 2.12（a）所示线性表在删除数据元素“郑彬”之后的状况。为了和指针型描述的线性链表相区别，这种用数组描述的链表称为静态链表，它适合在不设“指针”类型的高级程序设计语言中使用，虽然预先需要分配一个较大的空间，但在进行线性表的插入和删除操作时不需要移动元素，仅需修改指针，因此仍具有链式存储结构的主要优点。



图 2.12 静态链表示例

2.3.2 循环链表

循环链表（circular linked list）是一种表中最后一个结点的指针域指向头结点，整个链表形成一个环形的链式存储结构，因此从表中任一结点出发均可找到表中其他结点。如图 2.13 所示为单链的循环链表。循环链表的操作和线性链表基本一致，差别在于，判别链表中最后一个结点的条件不再是“后继是否为空”，而是“后继是否为头结点”。

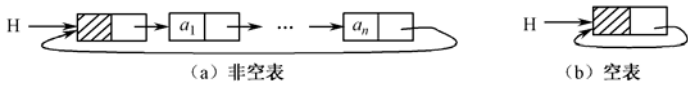


图 2.13 单链循环链表

2.3.3 双向链表

在需要频繁地同时访问前驱和后继结点时，可采用另一种链表结构，即双向链表。所谓双向链表，就是每个结点都有两个指针域，一个指向直接后继结点，另一个指向直接前驱结点，图 2.14 给出了结点结构和表结构。



图 2.14 双向链表示例

注意：① 双向链表由头指针唯一地确定；② 带头结点的双链表的某些运算变得方便；③ 将头结点和尾结点链接起来，即为双向循环链表。

双向链表的类定义见程序 2-16。

程序 2-16 双向链表的类定义

```
#include "LinearList.h"
template<class T>
class DoubleList;
template<class T>
class DoubleNode {
    friend class DoubleList<T>;
private:
    T data;
    DoubleNode<T> *prior, *next;
};
template<class T>
class DoubleList: public LinearList<T> //表示从 LinearList<T>派生
{
public:
    DoubleList( ) { LeftEnd=0; };
    virtual ~DoubleList( );
    virtual int Length( ) const;
```

```

virtual bool Find(int k, T& x) const;
virtual int Search(const T& x) const;
virtual DoubleNode<T> *GetNode(long i); //返回第 i 个结点
virtual bool Delete(int k, T& x);
virtual bool Insert(int k, const T& x);
virtual void Output(ostream& out) const;
virtual bool Get(DoubleNode<T> *node, T &x); //获取结点值
virtual bool Set(DoubleNode<T> *node, const T &x); //设置结点值
private:
    DoubleNode<T> *first, *last;
    //first>prior 指向最右边结点的指针 (即 first)
    //last>next 指向最左边结点的指针 (即 last)
};

```

在双向链表中，若  $p$  为指向表中某结点的指针，则显然有  $p \rightarrow \text{next} \rightarrow \text{prior} = p \rightarrow \text{prior} \rightarrow \text{next} = p$ 。对双向链表进行插入和删除时需同时修改两个方向的指针，它们的算法描述和线性表的操作有很大的不同。图 2.15 和图 2.16 分别显示了插入和删除结点时指针修改的情况。程序 2-17 和程序 2-18 分别对应插入和删除操作，两者时间复杂度均为  $O(n)$ 。

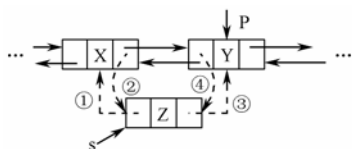


图 2.15 在双向链表中插入结点  
时指针的变化情况

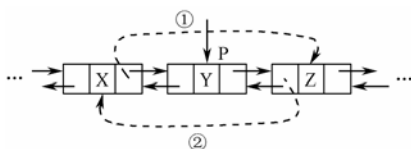


图 2.16 在双向链表中删除结点  
时指针的变化情况

### 程序 2-17 双向链表的插入算法

```

template<class T>
bool DoubleList<T>:: Insert(int k, const T& x)
{ //在第 k 个元素之后插入 x
    //如果不存在第 k 个元素，则引发异常 OutOfBounds
    //如果没有足够的空间，则传递异常 NoMem
    if (k<0) throw OutOfBounds ( );
    //p 最终将指向第 k 个结点
    DoubleNode<T> *p=first;
    //将 p 移动至第 k 个元素
    for (int index=1; index<=k && p; index++) p=p->link;
    if (k>0 &&!p) throw OutOfBounds ( ); //不存在第 k 个元素
    //插入
    LinkNode<T> *y=new LinkNode<T>; y->data=x;
    if (k) { //在 p 之后插入
        y->prior=p; y->next=p->next; p->next->prior=y; p->next=y;
    }
}

```



```

else { y->next=y; y->prior=y; first=y; } //作为第一个元素插入
return true;
}

```

## 程序 2-18 双向链表的删除算法

```

template<class T>
bool DoubleList<T>:: Delete(int k, T& x)
{
    //把第 k 个元素取至 x, 然后从链表中删除第 k 个元素
    //如果不存在第 k 个元素, 则引发异常 OutOfBounds
    if (k<1 || !first) throw OutOfBounds( ); //不存在第 k 个元素
    if (!p=GetElem(k)) return false;
    x=p->data; p->prior->next=p->next; p->next->prior=p->prior;
    delete(p);
    return true;
}

```

**【例 2-5】** 双向链表的定义及插入和删除操作算法。

在双向链表中, 每个结点的指针域有两个, 一个左指针 **llink**, 一个右指针 **rlink**, 至少一个数据域 **data**, 如图 2.17 所示。双向链表可以是循环的, 也可以不是循环的。

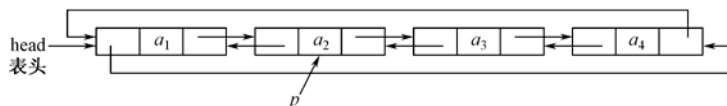


图 2.17 双向链表

对于双向链表 (循环链表), 若 **p** 指向表中任一结点, 则有  $p=p->llink->rlink=p->rlink->llink$ 。双向循环链表的定义及插入和删除一个结点的算法如下。

结点结构定义:

```

struct node2 { int data; node2 *llink; node2 *rlink; };
node2 *head; //双向环形链表的始指针

```

删除一个结点:

```

void delete(node2 *head, int x) //在以 head 为头指针的双向环形链表中, 删除结点 x
{
    node2 *p;
    p=head->rlink; //p 指向表中第一个结点
    while ((p!=head)&&(p->data!=x)) p=p->rlink; //查找结点 x
    if (p==head) cerr<<"无此结点"<<endl;
    else //找到结点 x
    {
        p->llink->rlink=p->rlink; //修改左结点的右指针
        p->rlink->llink=p->llink; //修改右结点的左指针
        delete p;
    }
}

```

插入一个结点：

```
void insert(node2 *head, int x, int y)
//在以 head 为头指针的双向循环链表中，在结点 x 的右端插入一个结点 y
{
    node2 *p;
    p=head->rlink;  //p 指向表中第一个结点
    while ((p!=head)&&(p->data!=x)) p=p->rlink;
    if (p==head) cerr<<"无此结点"<<endl;
    else { node2 *q=new node2; q->data=y; q->llink=p;
          q->rlink=p->rlink; q->rlink->llink=q; p->rlink=q; } //找到结点 x
}
```

2.3.4 顺序表和链表的比较

作为线性表的两种基本存储结构，顺序表和链表在存储与操作上各有所长，在实际应用中要根据具体问题的要求和性质来进行选择，表 2.1 给出了二者在主要方面的比较和对照。

表 2.1 顺序表和链表的对照表

		顺 序 表	链 表
基于空间考虑	分配方式	静态分配。在程序执行之前必须明确规定存储规模。若线性表长度 $n$ 变化较大，则存储规模难于预先确定，估计过大将造成空间浪费，估计太小又将使空间溢出机会增多	动态分配。只要内存空间中尚有空闲，就不会产生溢出。因此，当线性表的长度变化较大，难以估计其存储规模时，宜采用动态链表作为存储结构
	存储密度	为 1。当线性表的长度变化不大，易于事先确定其大小时，为了节约存储空间，宜采用顺序表作为存储结构	小于 1
基于时间考虑	存取方法	随机存取结构，对表中任一结点都可在 $O(1)$ 时间内直接取得。当线性表的操作主要是进行查找，很少进行插入和删除操作时，采用顺序表作为存储结构为宜	顺序存取结构，链表中的结点，需从头指针起顺着链扫描才能取得
	插入删除操作	在顺序表中进行插入和删除操作，平均要移动表中近一半的结点，尤其是当每个结点的信息量较大时，移动结点的时间开销相当可观	在链表中的任何位置上进行插入和删除操作，都只需要修改指针。对于频繁进行插入和删除操作的线性表，宜采用链表作为存储结构。若表的插入和删除操作主要发生在表的首尾两端，则采用尾指针表示的单循环链表为宜

2.4 线性表的应用——多项式相加与Josephus问题

2.4.1 多项式表示

多项式的数学表示式为  $A_n(x)=a_0+a_1 x+a_2 x^2+\cdots+a_n x^n$ ，在计算机内可很简单地通过线性表来表示  $A=(a_0, a_1, a_2, \cdots, a_n)$ ，每一项的指数  $i$  隐含在其系数  $a_i$  的序号里。然而，对于某些应用，多项式的次数可能很高且项数变化很大，例如， $B(x)=1+2x^{10000}+3x^{20000}$ ，就要用一个长度为 20001 的线性表来表示，而表中仅有 3 个非零元素，对于这种严重浪费内存空间的情况

应尽量避免。设一元  $n$  次多项式的通用形式为  $P_n(x)=p_1x^{e_1}+p_1x^{e_2}+\cdots+p_mx^{e_m}$ ，其中  $p_i$  是指数为  $e_i$  的项的非零系数，且满足  $0\leq e_1<e_2<\cdots<e_m=n$ ，上述情况可考虑采用一个长度为  $m$  且每个元素有两个数据项（系数项和指数项）的线性表  $P_n(x)$ :  $((p_1, e_1), (p_2, e_2), \cdots, (p_m, e_m))$  来存储，这样将大大节省空间。

在实际应用程序中，若只对多项式进行“求值”等不改变多项式的系数和指数的运算，则采用类似于顺序表的顺序存储结构即可，否则应采用链式存储表示。本节将主要讨论如何利用线性链表的基本操作来实现一元多项式的运算。例如，图 2.18 中的两个线性链表分别表示一元多项式  $A_{13}(x)=3+6x+9x^5+8x^{13}$  和一元多项式  $B_{10}(x)=7x^3-9x^5+12x^{10}$ ，每个结点表示多项式中的一项，并按照每项指数值的升序排列。

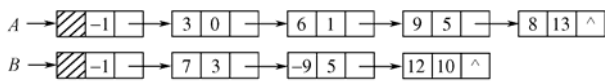


图 2.18 多项式表的单链存储结构

多项式相加的运算规则如下：先创建“和多项式  $C$ ”的头结点，并用指针  $p$ 、 $q$ 、 $r$  分别指向多项式  $A$ 、多项式  $B$  以及“和多项式  $C$ ”的头结点。 $p$ 、 $q$  从  $A$ 、 $B$  两个链表的第一个结点开始，比较它们所指向结点的指数值，复制指数值较小的结点并将其链接到  $r$  所指单链表的尾结点；若所指两个结点的指数值相等，且它们的系数值相加后不为零，则以该相加值和指数值为数据域值创建新的结点，并链接到  $r$  所指链表的尾结点；按照上述步骤进行并逐一移动  $p$ 、 $q$  指针，直至  $p$ 、 $q$  中有一个走到对应链表的尾结点为止，然后将未扫描完的多项式链表的结点复制并链接到  $r$  所指的“和多项式  $C$ ”的尾结点之后，则整个过程结束。由图 2.18 中的两个链表表示的多项式相加得到的“和多项式  $C$ ”链表如图 2.19 所示。线性链表类型适用于一般的线性表，而表示一元多项式的应该是有序链表。

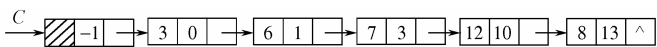


图 2.19 相加得到的“和多项式”

下面的程序 2-19 给出了多项式的类定义，程序 2-20 给出了多项式类的几种构造函数以及析构函数的定义。

程序 2-19 Polynomial 类定义

```
#include "LinearList.h"
#include "exception.h"
class Term{ //项的表示
public:
    float coef; //系数
    int exp; //指数
};
class Polynomial{
public:
    Polynomial(LinearListLink<Term>*list); //构造函数，用单链构造多项式类
    Polynomial( ); //默认构造函数
    Polynomial(Polynomial &); //用多项式类构造多项式类
```

```

virtual ~Polynomial();
//完成多项式相加运算
friend Polynomial & operator+(Polynomial &, Polynomial &);
//完成多项式相减运算
friend Polynomial & operator-(Polynomial &, Polynomial &);
//完成多项式相乘运算
friend Polynomial & operator*(Polynomial &, Polynomial &);
LinearListLink<Term>*GetPoly() { return poly; };
void SetPoly(LinearListLink<Term>*list) { poly=list; };
friend int Compare(int, int);
//比较两项指数, 相等则返回 0, 前者大则返回 1, 后者大则返回-1
private:
    LinearListLink<Term>*poly; //单链表结构, 数据类型是 Term
};

```

---

## 程序 2-20 Polynomial 类相关成员函数

---

```

#include "Polynomial.h"
#include "LinearListLink.h"
Polynomial::Polynomial(LinearListLink<Term>*list)
{
    if (list)
    {
        poly=new LinearListLink<Term>;
        LinkNode<Term>*node, *q, *p;
        Node=list->first;
        q=new LinkNode<Term>; q->data=node->data;
        node=node->link; p=q; poly->first=q;
        while (node)
        {
            q=new LinkNode<Term>; q->data=node->data;
            p->link=q; p=q; node=node->link; p->link=NULL; }
    }
}

Polynomial::Polynomial()
{
    Poly=new LinearListLink<Term>; poly=NULL; }

Polynomial::Polynomial(Polynomial &p)
{
    LinearListLink<Term>*list;
    List=p.poly;
    if (list)
    {
        poly=new LinearListLink<Term>;
        LinkNode<Term>*node, *q, *p;
        node=list->first;
        q=new LinkNode<Term>; q->data=node->data;
        node=node->link; p=q; poly->first=q;
    }
}

```

```

        while (node)
        {
            q=new LinkNode<Term>; q->data=node->data;
            p->link=q; p=q; node=node->link; }
            p->link=NULL;
        }
    }

Polynomial::~ ~ Polynomial( )
{
    //链表的析构函数，用于删除链表中的所有结点
    LinkNode<Term>*next;    //下一个结点
    if (poly)
    {
        while (poly->first)
        {
            next=poly->first->link; delete poly->first; poly->first=next; }
        }
    }
}

```

## 2.4.2 多项式相加

程序 2-21 给出多项式相加算法，请读者自行实现多项式的减法和乘法。

程序 2-21 多项式相加算法

```

Polynomial &operator +(Polynomial &a, Polynomial &b)
{
    //两个带头结点升序排序的多项式表头分别是 a 和 b，返回的是多项式
    //头指针 a（指向 a 的表头结点，不另外占存储空间，直接覆盖 a 和 b 链表）
    LinkNode<Term>*pa, *pb, *pc, *p;
    Term at, bt;
    pc=a.poly->Reset(0); p=b.poly->Reset(0); //p 定位于 b 的表头结点
    pa=a.poly->Reset(1); pb=b.poly->Reset(1); delete p;
    while (!a.poly->EndOfList() && !b.poly->EndOfList())
    {
        //都不是链表尾时循环
        at.exp=0; at.coef=0; bt.exp=0; bt.coef=0;
        a.poly->GetCurrent(at); //取当前结点值
        b.poly->GetCurrent(bt);
        switch(Compare(at.exp, bt.exp))
        {
            //pa 与 pb 的指数相等
            case 0:
                at.coef=at.coef+bt.coef; p=pb; pb=b.poly->Next(); delete p;
                if (at.coef<0.001) //相加为 0 时，删除该项
                {
                    p=pa; pa=a.poly->Next(); delete p; }
                else { pa->SetData(at); //修改结点的 data 值
                    pc->SetNext(pa); pc=pa;
                    pa=a.poly->Next(); //修改结点的 next 域
                }
            break;

```

```

case-1:
    pc->SetNext(pa); //修改结点的 next 域
    pc=pa;  pa=a.poly->Next();
    break;
case 1:
    pc->SetNext(pb); //修改结点的 next 域
    pc=pb;  pb=b.poly->Next();
}
}
if (!a.poly->EndOfList()) pc->SetNext(pa); //修改结点的 next 域
else pc->SetNext(pb);
return a;
}

```

**【例 2-6】** Josephus 问题，设有  $n$  个人围坐一圈。从第  $s$  人开始进行 1 到  $m$  报数，数到第  $m$  个人出列。然后，再从出列的下一个人重新开始报数，数到第  $m$  个人再出列。如此反复，直到所有的人全部出列为止。现要求按出列次序得出  $n$  个人的顺序表。现以  $n=8$ ,  $s=1$ ,  $m=4$  为例，用图说明报数出列的过程，如图 2.20 所示。图中用  $s_1$  作为浮动的指针，指向每次报数的起始位置。小圆圈内的数字为报数出列的人员，由图得到  $n$  个人按次序报数出列的人员顺序为④⑧⑤②①③⑦⑥。

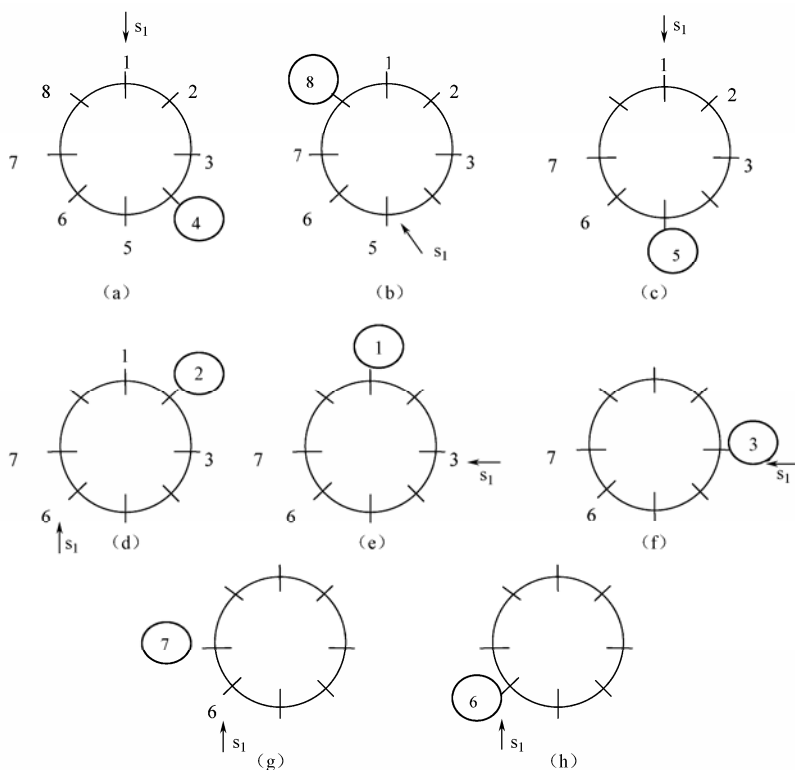


图 2.20 Josephus 问题

为了用计算机求解, 设想有一组  $1 \sim n$  的整数作为待报数的序列。将它们置于数组  $p$  中。当  $p[i]$  报数出列时, 为了节省空间, 可将  $p[i]$  从数组中删除并暂存在另一个存储单元  $w$  中, 且将  $p[i+1] \sim p[n]$  都向前移动一个位置。再将原保留在  $w$  中的  $p[i]$  值置于  $p[n]$  中, 如图 2.21 所示; 下次再对剩下的  $n-1$  个元素进行上述操作, 再将报数出列的元素置于  $p[n-1]$  中; ……; 如此反复, 直至只剩下一个元素时, 就将其保留在  $p[1]$  中原处。此时  $p$  中保留了报数出列的序列元素的逆序, 最后再将  $p$  数组中的元素逆转, 即得到了按次序报数出列人员的顺序表。

要完成上述运算还有一个关键问题需要解决, 即如何确定每次报数出列的位置? 由图 2.21 可见, 由于元素出列后, 后面的元素都要向前挪动一个位置, 因此, 本次报数的出列位置也就是下次报数的起始位置,  $s_1$  有双重含义。因为报数是首尾连续进行的, 所以可用取模的方法; 由这次报数的起始位置推算得到下次的报数位置, 即是本次报数的出列位置。设  $i$  为每次参加报数的人数, 则列出位置可由下式推算而得

$$s_1 \leftarrow (s_1 + m - 1) \text{ Mod } i$$

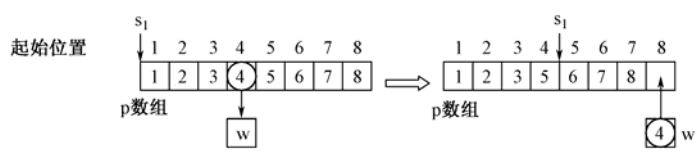


图 2.21 用计算机求解

Josephus 问题算法如下。①赋初值  $s_1 \leftarrow s$ 。② 报数出列, 循环  $i$  以  $-1$  为步长, 从  $n$  到 2 重复执行: (a) 求“出列”位置  $s_1 \leftarrow (s_1 + m - 1) \text{ Mod } i$ ; if  $s_1 = 0$  then  $s_1 \leftarrow i$ ; (b) 出列  $w \leftarrow p[i]$ ; (c) 元素前移: 循环  $j$  以 1 为步长, 从  $s_1$  到  $i-1$  重复执行:  $p[j] \leftarrow p[j+1]$ ; (d) 令出列元素于报数序列之尾  $p[i] \leftarrow w$ ; ③逆转出列的序列, 循环  $k$  以  $i$  为步长, 从  $i$  到  $\lfloor i/2 \rfloor$  重复执行: (a)  $w \leftarrow p[k]$ ; (b)  $p[k] \leftarrow p[n-k+1]$ ; (c)  $p[n-k+1] \leftarrow w$ 。(注:  $\lfloor i/2 \rfloor$  为不大于  $i/2$  的最大整数。)④输出已出列的序列, 算法结束 (具体程序请读者自己完成)。

# 本章总结

## 1. 学习要点

本章主要介绍线性表的概念及其逻辑结构形式定义; 顺序和链式两类存储结构的描述及实现方法; 在线性表的顺序和链式两类存储结构上, 如何进行数据元素的插入、删除、定位、表的归并、集合等基本运算, 线性表顺序与链式实现的时空性能比较, 以及用线性表表示一元多项式及其加法运算等内容。主要学习要点如下: ①线性表的概念、形式定义、线性结构的定义; ②顺序表与各种链表的优缺点及其适用场合; ③顺序表、单链表、循环链表的组织方法和实现插入、删除、定位等基本运算的算法; ④在各种链表上进行算法设计的基本技能; ⑤线性表的顺序实现与链接实现在空间性能和时间性能上的差异。

## 2. 基本要求

- (1) 深刻理解线性表及其运算和线性结构的概念;
- (2) 掌握线性表的逻辑结构定义, 并理解其形式化定义;

- (3) 弄清线性结构的基本特征;
- (4) 理解线性表中数据元素的“直接前驱”和“直接后继”的概念;
- (5) 弄清线性表 12 种基本运算的功能,特别是插入、删除及定位三种重要操作的定义;
- (6) 深刻领会线性表的顺序存储结构(顺序表)的基本思想;
- (7) 清楚顺序表方法的特点、类型定义、组成部分及作用、容量及表长的区别;
- (8) 灵活地应用线性表在顺序存储结构上的求址公式;
- (9) 熟练地掌握顺序表的插入、删除运算的算法和图示法表示;
- (10) 深刻领会线性表的链式存储结构基本思想;
- (11) 清楚单链表的类型定义、结点形式、逻辑关系、数据域和指针域的作用;
- (12) 正确区分头指针、头结点、首元结点,能熟练进行指针型变量操作;
- (13) 深刻理解单链表的插入、删除、定位三种基本运算的算法及其中所包含的指针运算;
- (14) 知道单向循环链表及双向循环链表的结点形式、定义、组织方法和特点;
- (15) 熟练掌握在单向或双向循环链表中的循环条件和插入、删除、定位算法;
- (16) 深刻理解线性表的顺序存储结构与链式存储结构的优缺点,并能灵活应用;
- (17) 知道线性表的顺序存储结构主要优点在于可随机访问表中任一结点,存储密度高,以及不易事先准确估计容量,插入、删除操作需移动大量元素等主要缺点;
- (18) 知道线性表的链式存储结构的主要优点是插入、删除操作方便快捷,不需事先估计链表的容量,以及存储密度低,不能随机访问链表中的任一结点等主要缺点;
- (19) 能独立地设计线性表(特别是单链表和双向循环链表)的综合应用算法。

### 3. 重点与难点

重点是:线性表结构的定义、特点,顺序表和单链表的组织方法和插入、删除、定位三种基本运算算法;难点是:在单链表和循环链表上进行综合应用算法设计。

## 习题 2

2-1 判断下列概念的正确性。

- ① 线性表在物理存储空间中一定是连续的。
- ② 链表的物理存储结构具有同链表一样的顺序。
- ③ 链表的删除算法很简单,因为当删去链表中某个结点后,计算机会自动地将后继的各个单元向前移动。
- ④ 使用双向链表存储数据,可提高查找(定位)的速度。

2-2 描述头指针、头结点、首元结点的区别。简述有序表的特性是什么?以及向量与有序表的异同点?头指针变量和头结点的作用?并比较顺序存储结构和链式存储结构的优缺点。

2-3 哪些链表可仅由一个尾指针来唯一确定,即从尾指针出发能访问到链表上任何一个结点。

2-4 设计一个删除向量中第  $i$  个元素的算法,并估算在等概率情况下算法的时间代价。

2-5 如果向量中的元素数据类型不同,这会有什么问题?

2-6 试设计算法,求循环链表中结点的个数并删除表中第一个结点。

2-7 已知线性表  $(a_1, a_2, \dots, a_n)$  中的元素值按递增有序排列,选用向量结构存放。试编写算法,删除线



性表中值介于  $c$  与  $d$  ( $c \leq d$ ) 之间的元素。

2-8 假设在长度大于 1 的循环链表中, 既无头结点也无头指针,  $p$  为指向链表中某个结点的指针, 试编写算法删除该结点的前驱结点。

2-9 设以双向链表为存储结构, 试编写算法实现线性表的去偶操作。

2-10 实现: ① 给出一个算法, 求线性链表里值为  $a$  的结点的地址, 这里设结点的值是整数。② 在线性链表中值为  $a$  的结点的后面插入一个值为  $b$  的结点, 结点的值是整数。

2-11 在双链表上实现线性表的下列基本运算: ① 初始化; ② 定位; ③ 插入; ④ 删除。

2-12 写出将链表  $L$  从某元素  $R$  处分成两个链表  $L$  和  $K$  的算法。  $R$  为  $K$  表的第一个元素。

2-13 若顺序存储的线性表中的一个元素长度为 20, 试计算其第 100 个元素的存储(相对)地址(设元素标号下限为 1)。

2-14 已知一个单链表如图 2.22 所示, 试给出一个算法将该线性表复制一个备份。

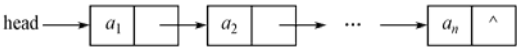


图 2.22 习题 2-14 的单链表

2-15 给定一个  $n$  项元素的线性表  $V$ , 写一个算法, 将元素排列的次序颠倒过来。要求占用原来的空间, 并用顺序表和单链表两种方法表示(用最少的附加空间来完成)。

2-16 线性表  $A$  和  $B$  中的元素都是字符, 它们都用向量结构存放, 试编写算法判断  $B$  是否为  $A$  的子序列。例如,  $A=\text{Search}$ ,  $B=\text{arc}$ , 则  $B$  为  $A$  的子序列。

2-17 假设  $x=(x_1, x_2, \dots, x_n)$  和  $y=(y_1, y_2, \dots, y_m)$  是两个线性链表, 试写一个将两个线性表合并为一个线性链表的算法。结点的物理位置不变, 这个新的线性链表为:

$x=(x_1, y_1, x_2, y_2, \dots, x_m, y_m, x_{m+1}, \dots, x_n)$ , 当  $n \geq m$  时;

$x=(x_1, y_2, x_2, y_2, \dots, x_n, y_n, y_{n+1}, \dots, y_m)$ , 当  $n < m$  时。

2-18 集合  $A$  与  $B$  的元素分别按递增有序的方式存放在两个循环链表中, 试编写算法, 从集合  $A$  中删除与集合  $B$  不相同的元素, 即求集合  $A$  和集合  $B$  的交集。

2-19 一个代数多项式由循环链表实现, 结构如图 2.23 所示, 编写算法, 按奇次项和偶次项把原链表拆成两个单链表, 要求不另开辟空间, 拆成的链表结点由原链表组成。

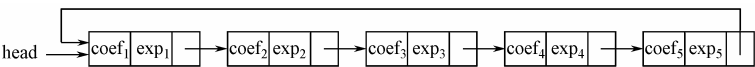


图 2.23 习题 2-19 的表结构

2-20 一个代数多项式由循环链表存放, 例如,  $P(x)=8x^5+3x^2+4$  的循环链表结构如图 2.24 所示, 结点由系数、指数和链指针三个域构成, 试编写一个对代数多项式求导数的算法。

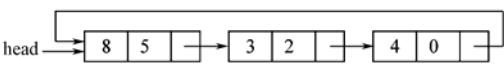


图 2.24 习题 2-20 的表结构

2-21 设  $A=(a_1, a_2, \dots, a_n)$  和  $B=(b_1, b_2, \dots, b_m)$  是两个线性表, 其结点值均是整数。若  $n=m$  且  $a_i=b_i$  ( $1 \leq i \leq n$ ), 则  $A=B$ ; 若  $a_i=b_i$  ( $1 \leq i < j$ ), 而  $a_i < b_i$  ( $j < n \leq m$ ), 则称  $A < B$ ; 除此之外, 均称  $A > B$ 。试编写一个比较  $A$  和  $B$  的程序, 并输出 -1、0 或 +1, 以表示  $A < B$ 、 $A=B$  或  $A > B$ 。

2-22 对于数组  $\text{Polyn}$  所表示的多项式, 试编写两个过程: ① 实现两个多项式相乘, 并进行算法分析。② 求多项式在  $x=x_0$  时的值, 要求用最小的运算次数加以实现, 例如,  $A(x)=5x^{1000}+3$ ,  $B(x)=2x^5+7x^3+x^2+1$

数组 Polyn 如图 2.25 所示。

coef	5	3	2	7	1	1		.....	
exp	1000	0	5	3	2	0		.....	
	↑ $A(x)$		↑ $B(x)$						

图 2.25 习题 2-22 的图

2-23 假设 A 和 B 是两个有序的循环链表，Pa 和 Pb 分别指向两个表的头结点。试写出一个算法，将这两个表归并为一个有序的循环链表。

2-24 假设有一个单向循环链表，其结点有三个域 pre、data、next。其中 data 为数据域；next 为指针域，其值为后继结点的地址；pre 也为指针域，它的值为空（NULL）。试编写算法将此链表改为双向循环链表。

2-25 设线性表存放在向量 A[0..arrsize] 的前 elenum 个分量中，且递增有序。试编写一个算法，将 x 插入到线性表的适当位置上，以保持线性表的有序性，并分析算法的时间复杂度。

2-26 已知单链表 L 中的结点是按值非递减有序排列的。试写一个算法将值为 x 的结点插入表 L 中，使得 L 仍然有序。

2-27 已知 A、B 和 C 为三个元素值递增有序的线性表。现要求对表 A 作如下操作：删去那些既在表 B 中出现又在表 C 中出现的元素。试编写分别以两种存储结构（一种顺序的、一种链式的）实现上述操作的算法，并分析每个算法的时间复杂度。

2-28 假设分别以两个元素值递增有序的线性表 A、B 表示两个集合（即同一个线性表中的元素各不相同）。现要求构成一个新的线性表 C。C 表示集合 A 与 B 的交，且 C 中元素也递增有序。试分别以顺序表和单链表为存储结构，编写实现上述运算的算法。

2-29 完成下列算法：

① list 是一个线性表，p、q、r 是 node\* 型的位置变量，确定函数 first、end 和 next 在下列程序中的执行次数（它们应该是表长 n 的函数）：

```
p=first(list);
while (p!=end(list))
{
    q=p;
    while (q!=end(list))
    {
        q=next(q, list);
        r=first(list);
        while (r!=q) r=next(r, list);
        p=next(p, list);
    }
}
```

② 下列程序试图消除在线性表 list 中出现的所有 x：

```
void delete(int x, node *list);
{
    node *p;
    p=first(list);
    while (p!=end(list))
    {
        if (retrieve(p, list)==x) delete(p, list);
        p=next(p, list);
    }
}
```

问该程序是否能在所有情况下完成预期的工作？若不能，则修改它。

③ 写一个合并两个已排序线性表的算法。

④ 写一个交换单向链表中位置  $p$  和  $\text{next}(p)$  的元素的程序。

2-30 试设计一个循环表，使表的每个结点只包含一个链域，而又能够有效地对它进行两个方面的查找。试编写一个程序，实现从表的一端开始，按顺时针方向访问这种表的每个结点。

2-31 ① 已知一个循环表如图 2.26 所示。试编写一个程序，将所有箭头方向取反。

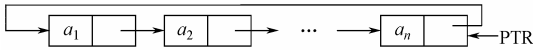


图 2.26 习题 2-31 的循环表

② 已知图 2.27 所示的双向循环链表  $L=(a, b, c, d)$ 。请写出将该表转换为  $L=(b, a, c, d)$  的简单操作。

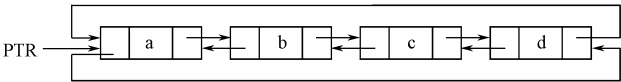


图 2.27 习题 2-31 的双向环形链表

2-32 写一个算法，将一个单向链表拆成两个循环链表，并将每个循环链表的长度存入表头结点的数域中。拆分规则如下：第一个循环链表包含原单向链表的第 1, 3, 5 ... 个结点，而第二个循环链表包含单向链表的第 2, 4, 6... 个结点。

2-33 分别以下述两种要求编写算法，将两个有序的线性表合并成一个有序的线性表，并分析比较这两个算法在时间和空间方面的代价。该线性表按顺序分配方式存放。① 允许另外开辟一个附加空间，以存放合并后的线性表。② 要求用最少的附加空间。此时，假设这两个有序的线性表在相邻的存储空间  $A(1:m)$  和  $A(m+1:n)$  中。

2-34 某百货公司仓库中有一批电视机，按其价格从低到高的次序构成一个循环链表。每个结点有价格、数量和链指针三个域，现新到  $m$  台价格为  $h$  的电视机，试编写算法修改原链表。

2-35 假设民航公司有一个自动预订飞机票的系统。该系统有一张用双重链表表示的乘客表。表中结点按乘客的姓氏字母顺序相链。下面是一张某个时刻的乘客表。试为该系统写出一个任一乘客要订票或退票时修改乘客表的算法。

序号	data	llink	rlink
1	LI	6	5
2	CHANG	4	9
3	WANG	5	7
4	BAI	0	2
5	MA	1	3
6	DU	8	1
7	XIA	3	0
8	DING	9	6
9	CHEN	2	8

2-36 (Josephus 环) 任意给正整数  $n, k$ ，按下述方法可得排列 1, 2, ...,  $n$  的一个置换：将数字 1, 2, ...,  $n$  环形排列（如图 2.28 所示），按顺时针方向从 1 开始计算，计满  $k$  时输出该位置上的数字（并从环中删去该数字），然后从下一个数字开始继续计算，直到环中所有数字均被输出为止。例如，当  $n=10, k=3$  时，输出的置换是 3, 6, 9, 2, 7, 1, 8, 5, 10, 4。试写一算法，对输入的任何正整数  $n, k$  输出相应的置换。



图 2.28 习题 2-36 的图

2-37 试用单向链表和循环链表的结构写出求解 Josephus 问题的算法。

2-38 假设有两个按元素值递增有序排列的线性表 A 和 B，均采用单链表作为存储结构。编写算法，将 A 表和 B 表合并成一个按元素值递减有序（即非递增有序，允许值相同）排列的线性表 C，并要求利用原表（即 A 表和 B 表的）的结点空间存放表 C。

2-39 已知一单链表中的数据元素含有三类字符（即字母字符、数字字符和其他字符）。试编写算法，构造三个循环链表，使每个循环链表中只含同一类的字符，且利用原表中的结点空间作为这三个表的结点空间（头结点可另辟空间）。

2-40 设有一个双链表，每个结点中除有 prior、data 和 next 三个域外，还有一个访问频度域 freq，在链表被起用之前，其值均初始化为零。每当在双链表上进行一次 Locate(L, x)运算时，令元素值为 x 的结点中 freq 域的值增 1，并使此链表中结点保持按访问频度递减的顺序排列，以便使频繁访问的结点总是靠近表头，试编写符合上述要求的 Locate 运算的算法。

2-41 实现以下操作：① A 和 B 两个仓库都存放有各种牌号的收录机。试设计一个以简单链表为数据结构的入库、出库的账目来往程序，要求标明价格、台数、牌号（A 库出，则 B 库入；否则相反）。② 试列举几个可应用线性表的信息管理系统的实例，一个使用向量存储结构，一个使用链式存储结构，画出系统数据结构示意图，并写出插入和删除操作算法。

2-42 为了用链表结构表示一个二进制整数，可定义链表单元的 celltype 为：

```
struct celltype
{
    int bit;
    celltype *next;
};
```

对于一个二进制整数  $b_1b_2\cdots b_n$ （其中每一位  $b_i$  为 0 或 1），可用一个表  $b_1, b_2, \cdots, b_n$  来表示，这个表是由 celltype 类型的单元链接而成的。利用这样的数据结构，编写一个将二进制整数加 1 的过程 increment(bnumber)，其中参数 bnumber 是一个指针，它指向表示该二进制整数的链表（提示：increment 写成递归过程）。

2-43 给定一个双链线性表，Left 和 Right 分别是指向表的左端和右端结点指针变量。设 x 是一个指针，试设计一个插入结点于 x 指向的结点的右边的算法。

2-44 试设计一个管理结点的系统（结点值为整数，结点总数不超过 1000），要求做到：每当程序中需要新结点时，可向该系统申请一个结点的存区；而当程序中不再使用该结点时，可把该结点的存区返回给此系统。设计内容包括：① 简述设计思想；② 说明适当的数组，并对它进行初始化；③ 写出向该系统申请一个结点存区的函数过程；④ 写出把结点存区返回给此系统的过程。

2-45 假设线性表 A 的结点是整数，试编写一个把 A 拆成 B、C 两个线性表的算法，使得 A 中大于零的结点存放在 B 中，而小于零的结点放在 C 中。

2-46 实现以下算法：① 为循环链表编写一个以逆转方向连接的算法；② 给出算法，求出循环链表中结点的个数；③ 写出一个算法，删除带头结点的循环链表的第一个结点；④ 用循环单链表编制一个飞机订票与退票的程序；⑤ 假设 a 和 b 是两个有序的循环链表，pa 和 pb 分别指向两个表的表头结点，试写出一个将两个表合并为一个有序的循环链表的过程。

2-47 已知一个单链表，试编写一个程序按递减次序打印各结点的数据域中的内容（提示：反复在链表中找出最大值的结点，打印该表直至表空）。

# 第3章 栈与队列

栈与队列是两种特殊的线性表，它们的逻辑结构与线性表的逻辑结构相同，但在运算操作方面较线性表有更多的限制，它们是重要的线性数据结构，通常被称为运算受限的线性表。在面向对象的程序设计中，它们是多型数据类型。

## 3.1 栈

### 3.1.1 栈的定义

栈（stack）是限制在表的一端进行插入和删除的线性表，又称为后进先出（last in first out）的线性表（简称 LIFO 结构）。进行插入和删除的一端是浮动端，称为栈顶（top），并用一个“栈顶指针”指示；另一端是固定端，称为栈底（bottom）。如图 3.1 所示，栈  $S=(k_1, k_2, \cdots, k_n)$ ，其中  $k_1$  为栈底元素， $k_n$  为栈顶元素。栈中元素按  $k_1, k_2, \cdots, k_n$  的次序进栈，退栈的第一个元素为栈顶元素。

下面再举两个关于栈的例子。例如，家里吃饭的碗通常在洗干净后一个一个地落在一起存放；在使用时，若一个一个地拿，一定最先拿走最上面的那个碗，而最后拿出最下面的那个碗。又如，建筑工地上使用的砖块从底往上一层一层地码放；在使用时，将从最上面一层一层地拿取。

在栈顶插入元素的操作通常称为入栈，删除栈顶元素的操作称为出栈。除此之外，栈的基本操作还包括栈的初始化、判空及取栈顶元素等。栈的抽象数据类型定义见 ADT 3-1。

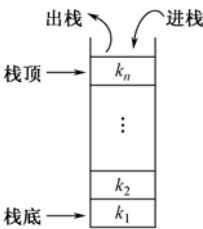


图 3.1 栈的示意图

ADT 3-1 栈的抽象数据类型描述

```
ADT Stack{
    数据集合:  $D=\{a_i \mid a_i \in \text{ElemSet}, i=1, 2, \cdots, n, n \geq 0\}$ 
    数据关系:  $R=\{\langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2, \cdots, n\}$ , 约定  $a_n$  端为栈顶,  $a_1$  端为栈底
    数据操作:
        Init_Stack(&S) //初始化栈
            输出: 构造一个空栈 S。
        Destroy_Stack(&S) //撤销栈
            输入: 栈 S。输出: 销毁栈 S。
        Clear_Stack(&S) //清空栈
            输入: 栈 S。输出: 将 S 清为空栈。
        Stack_Empty(S) //判断栈是否为空
            输入: 栈 S。输出: 若栈 S 为空栈, 则返回 TRUE; 否则返回 FALSE。
        Stack_Length(S) //求栈的长度
```

输入：栈 S。输出：返回 S 的元素个数，即栈的长度。

Get\_Top(S, &e) //取栈定元素

输入：栈 S 非空。输出：用 e 返回 S 的栈顶元素。

Push(&S, e) //栈定元素入栈

输入：栈 S。输出：插入元素 e 为新的栈顶元素。

Pop(&S, &e) //栈定元素出栈

输入：栈 S 非空。输出：删除 S 的栈顶元素，并用 e 返回其值。

Traverse\_Stack(S, visit( )) //遍历栈

输入：栈 S 非空。输出：从栈底到栈顶依次对 S 中的每个数据元素调用函数 visit( ) 进行访问。一旦 visit( ) 失败，则操作失败。

}ADT Stack

---

### 3.1.2 栈的抽象类

程序 3-1 给出了栈的抽象类定义，从面向对象的观点定义了栈的属性、方法，后面将介绍栈的顺序存储和链式存储所对应的类均是该抽象类的派生类。从基本操作上看，栈是线性表的子集，故应当是线性表的父类。这样，线性表类也可共享栈的方法。

程序 3-1 栈的抽象类

---

```
#include<iostream.h>
template<class T>
class Stack
{
public:
    Stack( );
    virtual ~Stack( );
    virtual bool IsEmpty( ) const=0; //判断栈是否为空
    virtual bool IsFull( ) const=0; //判断栈是否满
    virtual T& GetTop(void) const=0; //取栈顶元素
    virtual bool Push(const T &elem)=0; //元素 elem 进栈
    virtual bool Pop(T &x)=0; //出栈，并保存到 x 中
    virtual bool Pop( )=0; //出栈
    virtual void ClearStack( )=0; //清除栈中所有元素
    virtual void Output(ostream&out) const=0; //输出栈元素
};
```

---

### 3.1.3 栈的顺序存储结构

栈的顺序存储结构称为顺序栈，它利用一组地址连续的存储单元依次存放从栈底到栈顶的数据元素，并用一个变量记录栈顶元素的位置。通常采用数组来存放，习惯上将栈底放在数组下标小的那端。程序 3-2 给出了顺序栈的类描述。由于栈在使用过程中所需的最大空间很难估计。因此，一般来说，在初始化设空栈时不应限定栈的最大容量。

```
#include"exception.h"
#include"Stack.h"
template<class T>
class SeqStack:public Stack<T> //表示从 Stack<T>派生
{    //LIFO 对象
public:
    SeqStack(int MaxStackSize=10);    //构造函数
    ~SeqStack() { if (stack) delete [ ] stack; } //析构函数
    virtual bool IsEmpty() const { return top==-1; }
    //栈空返回 1, 否则返回 0
    virtual bool IsFull() const { return top==MaxSize-1; }
    //栈满返回 1, 否则返回 0
    T & GetTop() const;    //获得栈顶元素
    virtual bool Push(const T&x);    //元素 x 入栈
    virtual bool Pop(T& x);    //元素出栈, 并保存在 x 中
    virtual bool Pop();    //元素出栈
    virtual void ClearStack() { top=-1; };    //清除栈中所有元素
    virtual void Output(ostream& out) const;    //输出栈元素
private:
    int top;    //栈顶
    int MaxSize;    //最大的栈顶值
    T *stack;    //堆栈元素数组
};

template<class T>
SeqStack<T>::SeqStack(int MaxStackSize)
{    //Stack 类构造函数
    MaxSize=MaxStackSize; stack=new T[MaxStackSize];
    if (stack==NULL) throw NoMem(); top=-1;
}

template<class T>
T &SeqStack<T>::GetTop() const
{    if (IsEmpty()) throw OutOfBounds(); else return stack[top]; }
//返回栈顶元素

template<class T>
bool SeqStack<T>::Push(const T& x)
{    //添加元素 x
    if (IsFull()) throw OutOfBounds(); stack[++top]=x;
    return true;
}

template<class T>
bool SeqStack<T>::Pop(T& x)
{    //删除栈顶元素, 并将其送入 x 中
```

```

        if (IsEmpty()) throw OutOfBounds(); x=stack[top--];
        return true;
    }
    template<class T>
    bool SeqStack<T>::Pop()
    {   if (IsEmpty()) throw OutOfBounds(); top--; return true; }
    template<class T>
    void SeqStack<T>::Output(ostream& out) const
    {for (int i=0; i<=top; i++) out<<stack[i]<<" "; } //把表输送至输出流
    //重载<<
    template<class T>
    ostream& operator<<(ostream& out, const SeqStack<T>& x)
    {   x.Output(out);
        return out;
    }
}

```

类的数据成员 `stacksize` 指示栈的当前可使用的最大容量。栈的初始化操作为：按设定的初始分配量进行第一次存储分配，`top` 为栈顶指针，其初值指向栈底，值为-1，即 `top=-1` 可作为栈空的标记。每当插入新的栈顶元素时，指针 `top` 增 1；删除栈顶元素时，指针 `top` 减 1。因此，非空栈中的栈顶指针始终在栈顶第一个元素的位置上。图 3.2 展示了顺序栈中数据元素进栈和出栈时与栈指针之间的对应关系。

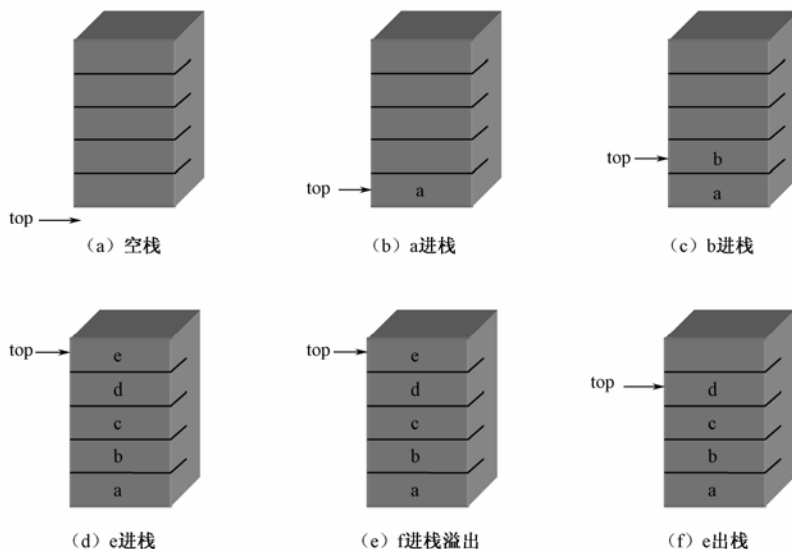


图 3.2 栈指针和栈中元素之间的关系

### 3.1.4 栈的链式存储结构

栈的链式表示，即链栈，如图 3.3 所示。由于栈的操作是线性表操作的特例，因此链栈可看成运算受限的单链表。其操作易于实现，因此不作详细讨论。其特点如下：① 链栈无栈满问题，空间可扩充；② 插入与删除仅在栈顶处执行；③ 链式栈的栈顶在链头；④ 适



合于多栈操作。程序 3-3 给出了链栈的类描述。

程序 3-3 链表形式的栈

```
#include"Stack.h"
template<class T>
class LinkedStack;
template<class T>
class Node{
    friend LinkedStack<T>;
private:
    T data;
    Node<T> *link;
};
template<class T>
class LinkedStack :public Stack<T>
{    //表示从 Stack<T>派生
public:
    LinkedStack( ) { top=0; }
    virtual ~LinkedStack( );
    virtual bool IsEmpty( ) const { return top==0; }
    virtual bool IsFull( ) const;
    virtual T& GetTop( ) const;
    virtual bool Push(const T& x);
    virtual bool Pop(T& x);
    virtual bool Pop( );
    virtual void ClearStack( );
private:
    Node<T> *top; //指向栈顶结点
}; //若 top 为栈顶指针，则空栈时 top=NULL
```

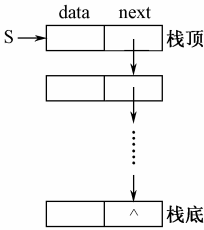


图 3.3 链栈示意图

### 3.2 栈的应用举例

“后进先出”特点使得栈在程序设计中的应用很广，下面给出一些栈应用的例子。

**【例 3-1】** 表达式求值。表达式求值是程序设计语言编译过程中的一个最基本问题。下面介绍一种简单直观、广为使用的“算符优先法”。

任何一个表达式都是由操作数（operand）、运算符（operator）和界限符（delimiter）组成的式子。一般地，操作数既可以是常数也可以是变量或常量。运算符从运算类型上可分为算术运算符、关系运算符和逻辑运算符三类。基本界限符包括左、右括号和表达式结束符等。为了简化问题，这里仅讨论只包含加、减、乘、除 4 种运算符的简单算术表达式的求值问题。运算符和界限符常被统称为算符。在运算规则中，表达式的算符之间有一定优先关系（见表 3.1），同时结合算术四则运算规则：① 先乘除，后加减；② 从左算到右；③ 先括号内，后括号外，则一个表达式的执行过程就能够被计算机正确地翻译并执行了。例如，对

于算术表达式  $8+12/3-2*4$  进行求值时, 该表达式的计算顺序应为  $8+12/3-2*4=8+4-2*4=12-2*4=12-8=4$ 。

表 3.1 运算符间的优先关系

$\theta_1 \backslash \theta_2$	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	,	=	
)	>	>	>	>	<	>	>
#	<	<	<	<	<		=

注意: 表 3.1 中的“#”是表达式的结束符, 为了算法的简捷, 可在表达式的最左边也虚设一个“#”构成整个表达式的一对括号。并且, 当“)”与“(”、“#”与“)”以及“(”与“#”这些之间无优先关系的算符对出现时, 可认为出现了语法错误。

在算法的实现过程中用到两个工作栈来分别存储算符和操作数或运算结果, 其基本思想是: 初始化操作数栈和算符栈, 表达式起始符“#”为算符栈的栈底元素; 自左向右扫描表达式, 若是操作数则进操作数栈, 若是算符则和算符栈的栈顶运算符比较优先权后进行相应的操作, 直至整个表达式求值完毕 (即 OPTR 栈的栈顶元素和当前读入的字符均为“#”)。程序 3-4 给出了算法的实现过程。

程序 3-4 表达式求值算法

```
#include"SeqStack.h"
const int Max=50;
OperandType EvaluateExpression ( )
{
    //算术表达式求值的算符优先算法。设 OPTR 和 OPND 分别为算符栈和操作数栈, OP 为算符集合
    char c, theta, x;
    dataType a, b;
    SeqStack<char>OPTR(Max); //建立运算符栈
    OPTR.Push('#'); //栈底放入“#”符号
    SeqStack<dataType>OPND(Max); //建立操作数栈
    cin>>c;
    while (c!='#' || OPTR.GetTop( )!='#')
    {
        if (!In(c, OP)) //不是运算符则进栈, In 为判断函数, OP 为运算符数组
        {
            OPND.Push(c); cin>>c;
        }
        else switch(Precede(OPTR.GetTop( ), c)) {
            //Precede 是判定运算符栈的栈顶运算符  $\theta_1$  与读入的运算符  $\theta_2$  之间优先关系的函数
            case '<':
                OPTR.Push(c); cin>>c; break; //栈顶元素优先权低
            case '=':
```

```

        OPTR.Pop(x); cin>>c; break; //脱括号并接收下一个字符
    case '>':
        OPTR.Pop(theta); OPND.Pop(b); //退栈并将运算结果入栈
        OPND.Pop(a); OPND.Push(Operate(a, theta, b));
        //Operate 为进行二元运算  $a\theta b$  的函数
        break;
    } //switch
} //while
return OPND.GetTop();
} //EvaluateExpression

```

算法中，**Precede** 函数用来判定运算符栈的栈顶运算符与读入的运算符之间的优先关系，**Operate** 函数进行二元运算。利用上述算法对表达式  $9/(1+2)$  求值的操作过程如下：

步骤	OPTR 栈	OPND 栈	输入字符	主要操作
1	#		9/(1+2) #	Push(OPND, '9')
2	#	9	/(1+2) #	Push(OPTR, '/')
3	#/	9	_(1+2) #	Push(OPTR, '(')
4	#/(	9	1+2) #	Push(OPND, '1')
5	#/(	9 1	+2) #	Push(OPTR, '+')
6	#/( +	9 1	) #	Push(OPND, '2')
7	#/( +	9 1 2	_)#	operate('1', '+', '2')
8	#/(	9 3	)#	Pop(OPTR) {消去一对括号}
9	#/	9 3	#	operate('9', '/', '3')
10	#	3	#	return(GetTop(OPND))

**【例3-2】** 背包问题。给定一组正整数权值  $w_1, w_2, w_3, \dots, w_n$ ，以及一个目标值  $t$ （整数）。试问：能否从这组权中选出若干个数，使它们的和等于  $t$ 。这就是所谓的“背包问题”。例如，当目标值  $t=10$ ，权值为 7, 5, 4, 4, 1 时，可选中第 2, 3, 5 这 3 个数作为解，这 3 个数之和为 10。如果设想权值分别是物品的重量，假设要用一个载重限度为  $t$  千克的背包携带物品，希望知道能否从这些物品中选出几件，使其总重量恰为目标值  $t$ 。

下面的程序给出了上述背包问题的一种递归解法。其中，函数 **knapsack** 的操作对象是由权值构成的数组 **weights[n]**；**knapsack(t, i)** 将回答是否能从权值 **weights[i]~weights[n]** 之间选出一些数，使其总和为  $t$ ，并在可能的情况下打印输出所选定的这些权值。在 3 种特殊情况下，**knapsack** 可立刻做出判断：当  $t=0$  时，有解，权的空集合就是一个解；当  $t<0$  时，无解，因为权值都是正整数；当  $t>0$ ，且  $i>n$  时也无解，这时所考虑的范围内无权值可选，当然不可能使其总和为  $t$ 。

当不是上述 3 种情况时，调用 **knapsack(t-w<sub>i</sub>, i+1)** 可验证是否存在包含  $w_i$  的解。如果调用的结果为 **true**，则说明问题有解，且  $w_i$  是解中选定的一个权值。因此，应将  $w_i$  打印输出；如果调用的结果为 **false**，则说明不存在包含  $w_i$  的解，那么解只可能选自  $w_{i+1} \sim w_n$  之间，因此调用 **knapsack(t, i+1)** 即可。

```

Bool knapsack(int target, int candidate)
{
    if (target==0) return true;
    else {
        if ((target<0)|| (candidate>n)) return false;
        else //考虑包括和不包括 candidate 的两种情况
        {
            if (knapsack(target - weights[candidate], candidate+1)
            {
                ...; //打印 weights[candidate]
                return true;
            }
            else //不可能有包括 candidate 的解
                return(knapsack(target, candidate+1))
        }
    }
}
}

```

**【例 3-3】** 地图四染色问题。“四染色”定理是计算机科学中著名定理之一，定理告诉我们，可用不多于 4 种颜色对地图进行染色（着色），使相邻的行政区域不重色。现在要用这个定理的结论，利用回溯算法对一张给定的地图染色。如图 3.4 所示，对每个行政区进行编号，1 色、2 色、3 色、4 色表示各个行政区域的颜色，称为色数。

从编号为 01 的区域开始逐一进行染色。对每个区域用色数 1、2、3、4 依次进行试探，并尽可能取小色数。若当前所取色数与周围已染色的区域不重色，则进栈记下该区域的色数；否则，依次用下一色数进行试探。若从 1 色到 4 色均与相邻某区域发生重色，则需退栈回溯，修改栈顶区域的色数。

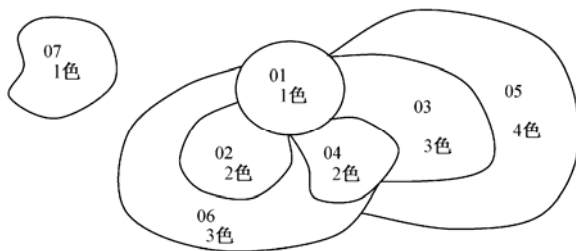


图 3.4 地图四染色问题

//二维数组 r[n][n] 设定待染色的 n 个区域的边界状况，s[n] 记载 n 个区域的染色结果

```

void mapcolour(int r[n][n], int s[n])
{
    s[0]=1; //01 号地区染 1 色
    int i=2, j=1, k; //i 为区域号, j 为染色号
    while (i<=n) //对第 i 号区域进行染色
    {
        while ((j<=4)&&(i<=n))
        {
            k=1; //k 指示已染色区域号
            while ((k<1)&&(s[k-1]* r[i-1][k-1]!=j))
                k++; //检测与已染色的相邻地区是否重色
            if (k<1) j++; //用 j+1 色继续试探
            else { s[i-1]=j; i++; j=1; }
        }
    }
}

```

```

        //不与相邻地区重色，进栈记下染色结果，继续对下一地区从 1 色起进行试探
    }
    if (j>4) { i--; j=s[i-1]+1; //变更栈顶区域的染色数 }
}
}

```

## 3.3 栈与递归

### 1. 递归

若在一个函数、过程或者数据结构定义的内部，直接（或间接）出现定义本身的应用，则称它们是递归的，或者是递归定义的。递归函数是指一个直接调用自己或通过一系列的调用语句间接调用自己的函数。递归是一种强有力的数学工具，它可使问题的描述和求解变得简捷和清晰。递归算法常常比非递归算法更容易设计，尤其是当问题本身或所涉及的数据结构是递归定义时，使用递归算法特别合适。

### 2. 递归算法的设计步骤

① 将规模较大的原问题分解为一个或多个规模更小、但具有类似于原问题特性的子问题，即较大的问题递归地用较小的子问题来描述，解原问题的方法同样可用来解这些子问题。

② 确定一个或多个无须分解、可直接求解的最小子问题（称为递归的终止条件）。例如，非负整数  $n$  的阶乘可递归定义为：

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

### 3. 栈在递归算法的内部实现中所起的作用

① 调用函数时，系统将为调用者构造一个由参数表和返回地址组成的活动记录，并将其压入到由系统提供的运行时刻栈的栈顶，然后将程序的控制权转移到被调函数。若被调函数有局部变量，则在运行时刻，在栈的栈顶也要为其分配相应的空间。因此，活动记录和这些局部变量形成了一个可供被调函数使用的活动结构。

② 被调函数执行完毕时，系统将运行时刻栈的栈顶的活动结构退栈，并根据退栈的活动结构中所保存的返回地址将程序的控制权转移给调用者继续执行。

## 3.4 队 列

### 3.4.1 队列的定义

队列（Queue）是只允许在一端进行插入、在另一端进行删除的运算受限的线性表。被允许删除的一端称为队头（front），被允许插入的一端称为队尾（rear）。队列亦称作先进先出（First In First Out）的线性表，简称为 FIFO 表。当队列中没有元素时，称为空队列。如图 3.5

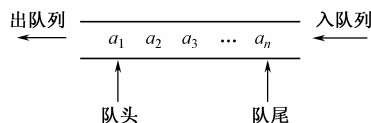


图 3.5 队列的示意图

所示，队列中的元素  $a_1, a_2, \dots, a_n$  必须按照从队头删除或从队尾加入的次序离开或进入队列。下面的程序分别给出了队列的 ADT 定义（见 ADT3-2）及其类描述（见程序 3-5）。

ADT3-2 队列的抽象数据类型定义

```
ADT Queue{
    数据集合:  $D=\{ a_i \mid a_i \in \text{ElemSet}, i=1, 2, \dots, n, n \geq 0 \}$ 
    数据关系:  $R=\{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2, \dots, n \}$ ，其中  $a_1$  为队头， $a_n$  为队尾
    数据操作:
        Init_Queue(&Q)    //初始化队列
            输出: 构造一个空队列 Q。
        Destroy_Queue(&Q) //撤销队列
            输入: 队列 Q。输出: 撤销队列 Q。
        Queue_Clear(&Q)   //清空队列
            输入: 队列 Q。输出: 队列 Q 变为空。
        Empty_Queue(Q)    //判断队列是否为空
            输入: 队列 Q。输出: 若队列 Q 为空队列，则返回 TRUE；否则返回 FALSE。
        Length_Queue(Q)   //求队列的长度;
            输入: 队列 Q。输出: 队列 Q 的元素个数，即队列的长度。
        Get_Head(Q, &e)   //取队列头元素
            输入: 队列 Q 为非空。输出: 用 e 返回队列 Q 的队头元素。
        En_Queue(&Q, e)   //入队
            输入: 队列 Q。输出: 插入元素 e 为队列 Q 的新的队尾元素。
        De_Queue(&Q, &e)  //出队
            输入: 队列 Q 为非空。输出: 删除队列 Q 的队头元素，并用 e 返回其值。
        Traverse_Queue(Q, visit( )) //遍历队列
            输入: Q 且非空。
            输出: 从队头到队尾依次对队列 Q 的每个数据元素调用函数 visit( ) 进行访问。
            一旦 visit( ) 失败，则操作失败。
    }ADT Queue
```

程序 3-5 队列的抽象类

```
template<class T>
class Queue
{
    public:
        Queue( );
        ~Queue( );
        virtual bool IsEmpty( ) const=0;
        virtual bool IsFull( ) const=0;
        virtual T& First( ) const=0;    //返回队首元素
        virtual T& Last( ) const=0;    //返回队尾元素
        virtual bool Insert(const T& x)=0;
        virtual bool Delete(T& x)=0;
```

```
virtual void visit()=0;
```

```
};
```

### 3.4.2 队列的顺序存储结构

队列的顺序存储结构称为顺序队列，顺序队列实际上是运算受限的顺序表。与顺序栈类似，在队列的顺序存储结构中，用一组地址连续的存储单元依次存放从队列头到队列尾的元素。指针 `front` 和 `rear` 被分别用来指示队列头元素和尾元素的位置。由于这两个指针可有多种指示队列状态的方法，为了便于描述，在本书中统一约定为：初始化建空队列时，`front=rear=0`；有新元素插入队尾时，`rear=rear+1`；从队列中删除头元素时，`front=front+1`；当队列非空时，头指针始终指向头元素，而尾指针始终指向队尾元素的下一个位置，如图 3.6 所示。

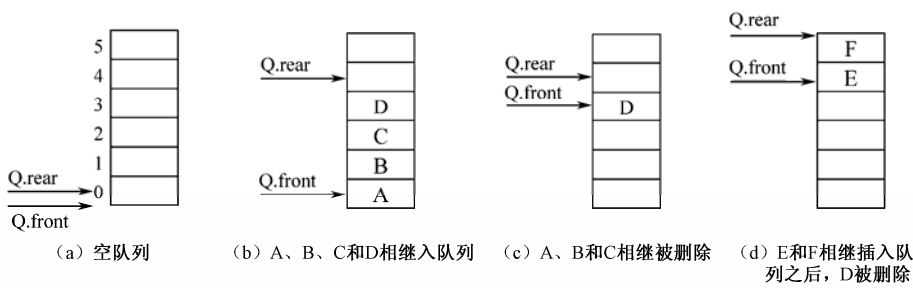


图 3.6 队列的顺序存储结构示意图

这种表示方法会遇到一个问题，假设队列的最大空间为 6，则在如图 3.6 (d) 所示的情况下，不能再向队列插入新的队尾元素，否则会导致数组越界，而此时队列的实际可用空间并未占满。对于这个问题，一个较好的解决办法是将顺序队列转换为一个循环空间，称为循环队列，如图 3.7 所示。图 3.6 所示的队列转换为循环队列后如图 3.8 所示，指针和队列元素之间关系不变。

从图 3.8 中的 (b) 和 (c) 可看出，用循环队列时会出现一个问题，即队满和队空时都有 `Q.front=Q.rear`，因此无法判断队列是“空”还是“满”。解决的办法之一就是：规定队列的最大长度始终小于存储空间大小，即留一个存储空间来区分队列空还是满。循环队列的类描述以及函数实现分别见程序 3-6 和程序 3-7。

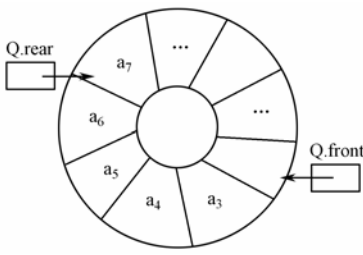


图 3.7 循环队列示意图

程序 3-6 循环队列的类描述

```
#include "Queue.h"
template<class T>
class SeqQueue:public Queue<T>
{    //FIFO 对象
public:
    SeqQueue(int MaxQueueSize=10);
    ~SeqQueue(); //{delete [ ] queue; }
    bool IsEmpty() const { return front==rear; }
```

```

bool IsFull() const { return(((rear+1)% MaxSize==front)? 1:0); }
T& First() const;    //返回队首元素
T& Last() const;     //返回队尾元素
bool Insert(const T& x);
bool Delete(T& x);
void visit();

private:
    int front;    //与第一个元素在反时针方向上相差一个位置
    int rear;     //指向最后一个元素
    int MaxSize;  //队列数组的大小
    T *queue;     //数组
};

```

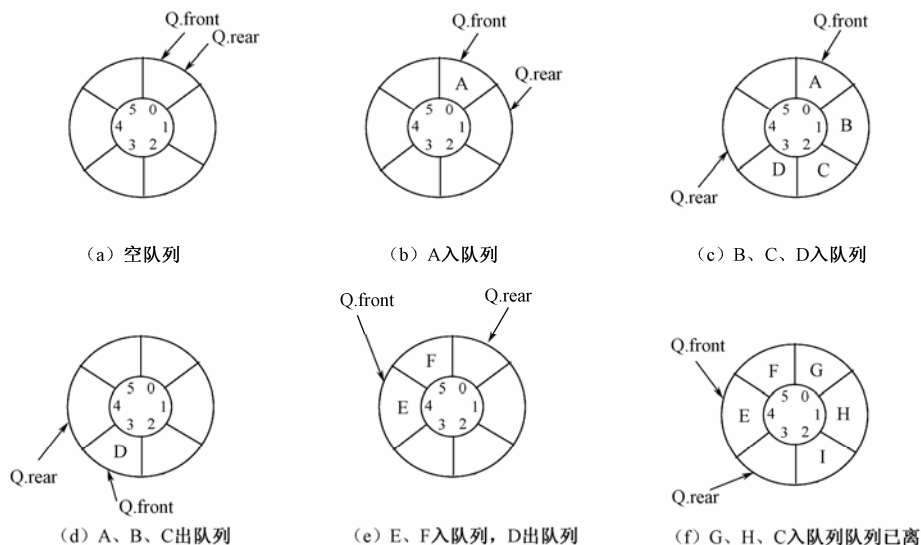


图 3.8 循环队列指针与元素关系示意图

### 程序 3-7 SeqQueue 的成员函数

```

template<class T>
SeqQueue<T>::SeqQueue(int MaxQueueSize)
{ //创建一个容量为 MaxQueueSize 的空队列
    MaxSize=MaxQueueSize+1; queue=new T[MaxSize]; front=rear=0;
}

template<class T>
SeqQueue<T>::~~SeqQueue()
{ if (queue) { delete [] queue; } }

template<class T>
T& SeqQueue<T>::First() const
{ //返回队列的第一个元素。如果队列为空，则引发异常 OutOfBounds
    if (IsEmpty()) throw OutOfBounds();
}

```



```

        return queue[(front+1)% MaxSize];
    }
    template<class T>
    T& SeqQueue<T>::Last( ) const
    {
        //返回队列的最后一个元素。如果队列为空，则引发异常 OutOfBounds
        if (IsEmpty( )) throw OutOfBounds( );
        return queue[rear];
    }
    template<class T>
    bool SeqQueue<T>::Insert(const T& x)
    {
        //把 x 添加到队列的尾部。如果队列满，则引发异常 NoMem
        if (IsFull( )) throw NoMem( );
        rear=(rear+1)% MaxSize; queue[rear]=x;
        return true;
    }
    template<class T>
    bool SeqQueue<T>::Delete(T& x)
    {
        //删除第一个元素，并将其送入 x。如果队列为空，则引发异常 OutOfBounds
        if (IsEmpty( )) throw OutOfBounds( );
        front=(front+1)% MaxSize; x=queue[front];
        return true;
    }
    template<class T>
    void SeqQueue<T>::visit( )
    {
        for (int i=front+1; i<=rear; i++) { cout<<queue[i]<<" "; }
        cout<<endl;
    }
}

```

**【例 3-4】** 在操作系统中，为了充分发挥计算机的效率，必须处理好中央处理机和外围设备在速度上不匹配的问题。信息的输入/输出都在内存中进行，需要设置输入/输出缓冲区，采用成组传送的办法，这样可实现主机与外部设备，以及外部设备与外部设备间的并行操作。在缓冲区内所有的信息都要排队，按照“先来先服务”的原则处理。因此，缓冲区实际上是一个队列的结构。下面模拟上述缓冲区的操作，当一个设备向缓冲区发送信息时，先要检查缓冲区中是否有空间，有，则送入缓冲区；无，则等待。当另一个设备接收缓冲区发出的信息时，先要检查缓冲区中是否有信息，有，则接收；无，则显示适当信息。

该程序应完成下述操作。

- ① 建立一个长度适当的队列作为缓冲器，用此缓冲器存放一组信息。
- ② 该程序对如下命令做出响应：S 发送信息，R 接收信息，E 程序结束。
- ③ 当发出 S 命令时，该程序检测队列中的空位，如果有可用的空间，则为要传送到队列中的信息显示提示；如果无可用空间，则显示“等待”信息。
- ④ 当发出 R 命令时，程序检测队列中的信息，并从队列中移出该信息同时显示出来；如果检测到队列中无信息，则显示“缓冲器空”提示。

⑤ 当发出 E 命令时，操作结束，退出本程序。

程序结构分为三层：上层为解释命令，中层包括执行上述命令的操作模块，下层为几个服务模块，包括完成插入或移出运算的子程序及一个送回当前队列长度的子程序。一般发送和接收的信息多为字符，用不定长字符 A 和 B 分别表示发送和接收的信息。因此，前面定义的队列 qu 中数组 q 的基类型应为 char \*型。

发送模块算法如下。

① 计算当前队列长度 l。

② 若队列有空 ( $l < m-1$ ) 则：(a) 要求用户输入 A；(b) 执行 `inq(qu, A)`。(A 插入队列中)，否则算法结束。

③ 算法结束。

接收模块算法如下。

① 计算队列长度 l。

② 若  $l > 0$ ，则执行 `qfdeq(qu, B)`。(读队头元素 B)，然后执行 `deq(qu)`，(删除 B)；否则 `write`(“缓冲器空”)。

③ 算法结束。

主模块除了识别用户命令，将控制转移到相当的操作模块外，还给出程序系统提示符“\*>”，初始化一个队列，包括规定队列的最大容量 m，并置队列为空： $f=r=1$ 。程序如下：

---

```
const m=4;
struct queue
{char *q[m]; int f, r; }
char *a, *b; queue qu; char c, ch; int l;

void buffer()
{   qu.f=0; qu.r=0;
    while ((c!='E')&&(c!='e'))
    {   cout<<"*>"; cin>>c; cout<<endl;
        switch (c)
        {   case 'S':
                case 's': send(); break;
                case 'R':
                case 'r': receive(); break;
                default: cout<<"输入错误! 请重新输入!"<<endl;
            }
        }
    }

void inq( )
{   if ((qu.r==qu.f-1) || ((qu.f==0)&&(qu.r==m-1))) cout<<"溢出。"<<endl;
    else {qu.q [qu.r]=a; if (qu.r==m-1) qu.r=0; else qu.r++; }
}
```

```

void qfdeq( )
{
    if (qu.f!=qu.r)
        {b=qu.q [qu.f]; if (qu.f==m-1) qu.f=0; else qu.f++; }
}

void len( )
{if (qu.r<qu.f) l=qu.r+m-qu.f;else l=qu.r-qu.f;}

void send( )
{
    ch=' ';
    while ((ch!='y')&&(ch!='Y'))
    {
        len( );
        if (l<m-1)
        {
            cout<<"请输入要发送的数据."<<endl; readln (a); //调用输入函数
            inq( );
        }
        else {cout<<"请等待!"<<endl; ch='N'; }
        if (ch!='N') {cout<<"要发送吗? "; cin>>ch; cout<<endl; }
    }
}

void receive( )
{
    len( );
    if (l>0) {qfdeq( ); cout<<"收到消息:"<<b<<endl; }
    else cout<<"无消息。"<<endl;
}

```

### 【例 3-5】 模拟打印机缓冲区。

在主机将数据输出到打印机中时，会出现主机速度与打印机的打印速度不匹配的问题。这时，主机就要停下来等待打印机。显然，这样会降低主机的使用效率。为此，人们采用了一种办法：为打印机设置一个打印数据缓冲区，当主机需要打印数据时，先将数据依次写入这个缓冲区中；写满后，主机转去做其他的事情，而打印机从缓冲区中按照先进先出的原则依次读取数据并打印。这样，既保证了打印数据的正确性，又提高了主机的使用效率。由此可见，打印机缓冲区实际上就是一个队列结构。具体程序请读者自己完成。

**【例 3-6】 舞伴问题。**假设在舞会上，男士和女士进入舞厅时，各自排成一队。跳舞开始时，依次从男队和女队的队头上各出一人配成舞伴。若两队初始人数不相同，则较长的那一队中未配对者等待下一轮舞曲，现要求通过算法来模拟上述舞伴配对问题。

由于先入队的男士或女士亦先出队配成舞伴，因此该问题具有典型的先进先出特性，可用队列作为算法的数据结构。在算法中，假设男士和女士的记录存放在一个数组中作为输入，然后依次扫描该数组的各元素，并根据性别来决定是进入男队还是女队。当这两个队列构造完成之后，依次将两队当前的队头元素出队来配成舞伴，直至某队列变空为止。此时，若某队中仍有等待配对者，算法输出此队列中等待者的人数及排在队头的等待者的名字，他

(或她)将是下一轮舞曲开始时第一个可获得舞伴的人。具体程序请读者自己完成。

### 3.4.3 队列的链式存储结构

链队列，即用链表表示的队列，它是限制仅在表头删除和表尾插入的单链表。如图 3.9 所示，有两个分别指向队头和队尾的头指针和尾指针。另外，在链表前还添加了一个头结点以使操作更加简便。空的链队列的判决条件为：头指针和尾指针均指向头结点，如图 3.10 所示。程序 3-8 和程序 3-9 分别给出了链队列的类定义和各种操作的函数实现。

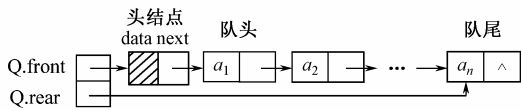


图 3.9 链队列示意图



图 3.10 空链队

程序 3-8 链表队列的类定义

```
#include "exception.h"
#include "Queue.h"
template<class T>
class LinkedQueue;
template<class T>
class Node{
    friend LinkedQueue<T>;
private:
    T data; Node<T> *link;
};

template<class T>
class LinkedQueue:public Queue<T>
{    //FIFO 对象
public:
    LinkedQueue( ) { front=rear=0; } //构造函数
    ~LinkedQueue( ); //析构函数
    bool IsEmpty( ) const { return((front)?false:true); }
    bool IsFull( ) const;
    T& First( ) const; //返回第一个元素
    T& Last( ) const; //返回最后一个元素
    bool Insert(const T&x);
    bool Delete(T&x);
    void visit( );
private:
    Node<T> *front; //指向队头结点
    Node<T> *rear; //指向队尾结点
};
```

```

template<class T>
LinkedList<T>::~~LinkedList( )
{
    //队列析构函数，删除所有结点
    Node<T> *next;
    while (front) { next=front->link; delete front; front=next; }
}

template<class T>
bool LinkedList<T>::IsFull( ) const
{
    //判断队列是否已满
    Node<T> *p;
    try { p=new Node<T>; delete p; return false; }
    catch(NoMem) { return true; }
}

template<class T>
T& LinkedList<T>::First( ) const
{
    //返回队列的第一个元素
    //如果队列为空，则引发异常 OutOfBounds
    if (IsEmpty( )) throw OutOfBounds( ); return front->data;
}

template<class T>
T& LinkedList<T>::Last( ) const
{
    //返回队列的最后一个元素。如果队列为空，则引发异常 OutOfBounds
    if (IsEmpty( )) throw OutOfBounds( ); return rear->data;
}

template<class T>
bool LinkedList<T>::Insert(const T& x)
{
    //把 x 添加到队列的尾部
    //不捕获可能由 new 引发的 NoMem 异常，为新元素创建链表结点
    Node<T> *p=new Node<T>; p->data=x; p->link=0;
    //在队列尾部添加新结点
    if (front) rear->link=p; //队列不为空
    else front=p; //队列为空
    rear=p; return true;
}

template<class T>
bool LinkedList<T>::Delete(T& x)
{
    //删除第一个元素，并将其放入 x
    //如果队列为空，则引发异常 OutOfBounds
    if (IsEmpty( )) throw OutOfBounds( );
    x=front->data; //保存第一个结点中的元素
    Node<T> *p=front; front=front->link; delete p; //删除第一个结点
    return true;
}

```

```

}

template<class T>
void LinkQueue<T>::visit( )
{
    Node<T> *next;    next=front;
    while (next) {    cout<<next->data<<" ";    next=next->link; }    cout<<endl;
}

```

---

有一点需要注意，在一般情况下，删除队列头元素时，仅需修改头结点中的指针，但当队列中最后一个元素被删后，队列尾指针也丢失了，因此需要对队尾指针重新赋值，即指向头结点。

## 本章总结

### 1. 学习要点

本章主要介绍了两类特殊的线性表——栈和队列的概念，以及顺序和链式两类存储方式的描述和实现方法、栈和队列的相关应用等，主要学习要点如下：① 栈和队列的特点及其与线性表的异同；② 顺序栈与链栈的组织方法，基本运算的实现；③ 入栈及出栈运算的算法，递归过程的模拟；④ 循环队列及链队列的组织方法，队满、队空的判断条件及其描述和算法；⑤ 栈与队列综合应用算法的设计表达式求值。

### 2. 基本要求

- (1) 深刻领会栈的概念、实现和基本运算；
- (2) 清楚栈的逻辑结构、基本特点（FILO），实质上，栈是一种特殊的线性表；
- (3) 弄清栈的顺序实现的基本思想及其基本运算，以及“上溢”和“下溢”的概念；
- (4) 掌握判断顺序栈满和栈空的条件及在顺序栈上实现入栈和出栈运算的算法；
- (5) 清楚链栈的定义、结点形式实现方法以及与顺序栈的不同特点；
- (6) 掌握在链栈上进行入栈和出栈运算的算法及判断栈空和栈满的条件；
- (7) 会在顺序栈和链栈上进行综合应用程序设计及递归过程模拟；
- (8) 深刻领会队列的概念、实现和基本运算；
- (9) 知道队列的逻辑结构、基本特点（FIFO），以及其与线性表和栈的异同；
- (10) 清楚队列实质上也是一种特殊的线性表；
- (11) 知道循环队列的结点形式、实现方法、判断队空和队满的条件；
- (12) 掌握在循环队列上进行入队列和出队列运算的基本算法；
- (13) 清楚链队列的类型定义、结点形式、组织方法、判断队空和队满的条件；
- (14) 掌握在链队列上进行入队列和出队列运算的基本算法；
- (15) 能在循环队列和链队列上进行综合应用程序设计；
- (16) 掌握递归过程的程序设计方法。

### 3. 重点及难点

重点是：栈与队列的特点；在顺序栈和链栈上实现基本运算；在循环队列和链队列上

实现基本运算的算法和栈与队列的综合应用程序设计。难点是：循环队列的队空、队满条件；用栈和队列进行综合应用程序设计，以及递归过程设计和模拟。

### 习题 3

- 3-1 说明栈和队列与线性表的异同点。
- 3-2 说明下述运算的结果（S 为栈名）：① $\text{pop}(\text{push}(\text{S}, \text{A}))$ ；② $\text{push}(\text{S}, \text{pop}(\text{S}))$ ；③ $\text{push}(\text{S}, \text{pop}(\text{push}(\text{S}, \text{B})))$ 。
- 3-3 若  $\text{pop}(\text{S})=\text{A}$ ，试说明下式是否具有相同运算结果，即是否有等式
- $$\text{pop}(\text{push}(\text{S}, \text{A}))=\text{push}(\text{S}, \text{pop}(\text{S}))$$
- 3-4 写一个算法，借助栈将一个单链表倒置。
- 3-5 从现实生活中举例说明：栈的特征、队列的特征，栈的“上溢”、“下溢”现象，以及顺序队列的“假溢出”现象。
- 3-6 试写出一个利用栈将中缀表达式（设只包含特有运算符： $+$ 、 $-$ 、 $\times$ 、 $\div$ 和括号“ $($ ”、“ $)$ ”），转换成后缀表达式的算法。
- 3-7 对于一个适当大小的栈，设输入项序列为 A、B、C、D、E，为得到下列的处理序列，需要做什么样的运算序列（由 PUSH、POP 组成）？如果以下某处理序列得不到，试说明理由：① A、B、C、D、E；② B、C、D、E、A；③ E、A、B、C、D；④ E、D、C、B、A。
- 3-8 利用一个栈求下列各表达式的值，对于每一个表达式，栈必须拥有多少项（假定栈只容纳运算符）？①  $\text{A}-\text{B}$ ；②  $(\text{A}-\text{B})-\text{C}$ ；③  $\text{A}-(\text{B}-\text{C})$ ；④  $(\text{A}-\text{B})-(\text{C}-\text{D})$ ；⑤  $(\text{CA}-\text{B})-\text{C}-(\text{D}-(\text{E}-\text{F}))$ 。
- 3-9 设计算法判断一个算术表达式的括号是否正确配对。（提示：对表达式进行扫描，遇“ $($ ”则进栈，遇“ $)$ ”则退栈顶的“ $($ ”。表达式被扫描完毕，栈应为空。）
- 3-10 设有两个栈，共享空间  $\text{Space}[1..n]$ ，如图 3.11 所示。试编写一个对任一栈进行进栈和退栈运算的程序过程或函数： $\text{push}(x, i)$ 和  $\text{pop}(i)$ ， $i=1, 2$ 。这里， $i=1$  表示左边的栈， $i=2$  表示右边的栈。要求：算法只有在整个空间  $[1..n]$ 全满时，才会产生溢出。



图 3.11 习题 3-10 的图

- 3-11 编写程序：实现检验第  $i$  个栈是否栈满的算法  $\text{stackfull}(i)$ 。
- 3-12 用链表存放着  $n$  个字符，试用算法判断读字符串是否为中心对称。例如， $\text{abccba}$ ， $\text{abcba}$  都是中心对称的字符串。要求：用尽可能少的时间完成判断（提示：将一半的字符先依次进栈）。
- 3-13 试列举科学实践中应用栈和队列的例子，并给出它们的物理表示以及处理它们的算法。
- 3-14 对下列函数，画出调用  $\text{f}(5)$ 时引起的工作栈状态变化情况。

```
int f(int i)
{   if (n==1) return 10; else return(f(i-1)+2); }
```

- 3-15 写出链栈的取栈顶元素和判栈空的算法以及  $\text{push}$ ， $\text{pop}$ ， $\text{top}$ ， $\text{empty}$  和  $\text{makenull}$  的算法。
- 3-16 有字符串次序为  $3^*-\text{y}-\text{a}/\text{y} \uparrow 2$ 。试利用栈排出将次序改为  $3\text{y}-*\text{ay}2 \uparrow /-$ 的操作步骤。（提示：可用

X 代表在扫描该字符串过程中, 顺序取一个字符进栈的操作, 用 S 代表从栈中取出一个字符加入到新字符串尾的出栈操作。例如, ABC 变为 BCA, 则操作步骤为 XXSXSS。)

3-17 假定以单链表结构存储整数序列  $F=a_1a_2\cdots a_n$  ( $a_1\leq a_2\leq a_3\cdots\leq a_k$ ) 和空闲栈 V, 它们的始指针分别为  $P_i$  和  $P_v$ 。试设计一程序, 对于存在  $a_i=a_{i+1}$  的一切  $i$  ( $1\leq i\leq k-1$ ), 从链表中删除  $a_{i+1}$ , 且将空出来的存储单元归还到空闲栈去。

3-18 现有中缀表达式  $E=((100-4)\div 3+3\times(36-7))\times 2$ , 试写出与 E 等价的后缀表达式, 并用栈来模拟表达式的转换过程, 画出转换过程中栈内容的变化图。

3-19 表达式除了中缀和后缀形式之外, 还有一种前缀表达式, 即书写时, 运算符置于操作数的前面。

例如,	中缀表达式	前缀表达式
	A+B	+AB
	A+B*C	+A*BC
	A*(B+C)	*A+BC
	A*B+C	+*ABC
	A+B*C+D-E*F	-++A*BCD*EF
	(A+B)*(C+D-E)*F	**+AB-CDEF

① 设计一个将中缀表达式转换成前缀表达式的算法 (提示: 中缀表达式以 @ 作为结束符, 而前缀表达式以 @ 开始)。

② 另设计一个计算前缀表达式的算法 (提示: 自右至左扫描前缀表达式, 并假设前缀表达式以 @ 开始)。

③ 设计从前缀表达式转换成后缀表达式的算法。

3-20 如图 3.12 所示为一个铁路调度站, 为一栈式结构。右边轨道上排列有编号为 1、2、3 的三节车厢。每节车厢可被带进站, 并可在任意时刻拖走。若将车厢 1、2、3 依次带进, 则从在边轨道上出站的可能顺序有多少种? 请把它们具体写出来。若有 1、2、3、4 共 4 节车厢在右边轨道上依次进栈, 则情况又将如何?

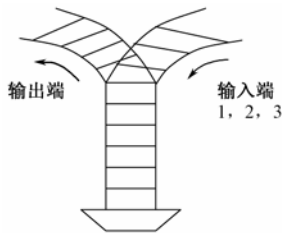


图 3.12 习题 3-20 的铁路调度站

3-21 将任意  $n$  个数从小到大链接起来, 存储区中的空闲单元链接成一个空闲存储栈, 按以下要求写出算法: ①查找链表中的任一个结点 (若有相同元素, 则给出所有相同元素的个数及它们的地址); ②增添一个结点, 并按大小次序链接; ③删除一个结点, 若有相同元素, 则全部删去, 并按其所占单元链入空闲存储栈。

3-22 利用两个栈 S1 和 S2 模拟一个队列, 并写出队列空、入队和出队的算法。

3-23 试列举信息管理系统中需要栈和队列操作的例子, 并画出它的物理表示和处理算法。

3-24 假设 P 是一个简单循环链表表头指针。将该链表作为队列, 试写出入队和出队算法。

3-25 习题 3-24 中, 链表若为堆栈, 试写出入栈和出栈算法。

3-26 双向队列的插入与删除可在任何一端进行。写出顺序存储表示时在双向队列的任一端插入和删除的算法过程。

3-27 试用布尔函数来表示循环队列为空或者满, 并应用这些布尔函数写出循环队列中插入结点和删除结点的过程。

3-28 线性表循环地存放在数组 C[n] 中, 将数组 C 看作一个循环队列, head 和 tail 是它的队头指针和队尾指针。①利用 head, tail 和 n 计算表的元素个数; ②写一个删去表中第 k 个元素的算法; ③写一个将元素 y 插入到表中第 k 个元素后面的算法。



- 3-29 循环队列的结构可以用循环链表来实现, 在这种结构下, 写出入队和出队操作的算法。
- 3-30 Cq[10]是一循环队列, 初始状态为 front=rear=0, 画出做完下列操作后队列的头尾指示器状态变化情况。若不能入队请指出其元素, 并说明理由。  
d, e, b, g, y 入队; b, e 出队; i, j, k, l, m 入队, b 出队; n, o, p, q, r 入队
- 3-31 设有一空队列, 它的起始地址  $q_0=5$ ,  $m=3$ , 试给出 head, tail 的初值以及下列每步运算之后的值: En\_queue En\_queue De\_queue En\_queue En\_queue En\_queue De\_queue。其中, 假定从空队列出或进入满队列, 都不改变 head, tail 的值 (进队元素长度为 1)。
- 3-32 对于循环队列, 试写出求队列长度的算法。
- 3-33 假设以带结点的循环链表表示队列, 并只设一个指针指向队尾元素结点 (注意, 不设头指针), 试编写相应的置空队列、入队列和出队列算法。
- 3-34 假设以数组 cycque[0..m-1]存放循环队列的元素, 同时设变量 rear 和 quelen 分别指示循环队列中队尾元素的位置和内含元素的个数。试给出此循环队列的队满条件, 并写出相应的入队列和出队列的算法。
- 3-35 如果队列中不设表头结点, 如图 3.13 所示, 并令 front=rear=null 表示队列空, 试写出算法, 实现队列的 5 个基本操作 (置空、入队、出队、判空和读队头)。



图 3.13 习题 3-35 的图

- 3-36 设有一个带表头的链式线性表, 表头地址为 head, 试给出相当于栈的插入操作和删除操作。
- 3-37 用循环数组实现队列时, 不设后端指针 rear, 而改设置计数器 count 用于记录队列中满结点的个数: ①写出实现队列的 5 个基本操作的算法; ②队列中能容纳元素的最多个数还是  $n-1$  吗?
- 3-38 阿克曼函数  $A(m, n)$  定义如下:
- $$A(m, n) = \begin{cases} n + 1 & \text{若 } m = 0 \\ A(m - 1) & \text{若 } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{其他} \end{cases}$$
- 编写一个计算这个函数的递归过程, 再编写一个计算阿克曼函数的非递归算法。
- 3-39 用栈的方法去掉下面过程中的递归调用:

```
int  comb(int n, int m) //设  $0 \leq m \leq n$  和  $n \geq 1$  时, 计算  $(n, m)$ 
{
    if ((n==1) || (m==0) || (m==n)) return 1;
    else return (comb(n-1, m) + comb(n-1, m-1));
}
```

## 第4章 串

现实世界中的实体有多种形式，如数值、文字符号序列、图形、图像、声音等，其中文字（符号）序列称为字符串，简称串。串是一种常用的数据结构，用于描述非数值的简单信息，它在现实世界中屡见不鲜，如人名、产品名、事物名称、车牌号、文献、源程序等。从数据元素间的逻辑关系看，串是线性表，但从操作方式与种类看，它与线性表有许多不同。在计算机中，串被认为是一种特殊的线性表——元素为文字符号的线性表。由于现实问题中经常使用串，所以对串应选择合适的存储结构，并提供灵活多样的基本操作。

### 4.1 串的逻辑结构

#### 4.1.1 基本概念

串（string）是由零个或多个字符构成的有限序列，一般记为  $s = 's_1s_2\cdots s_n'$  ( $n \geq 0$ )。其中， $s$  是串名，用单引号括起来的字符序列是串的值，但单引号本身并不属于串， $s_i$  ( $1 \leq i \leq n$ ) 为一个字符，字符是计算机可识别的任意符号（字母、数字或其他符号）。

串的长度：串中字符的数目  $n$ 。

空串：零个字符的串，其长度为零。

空白串：由一个或多个空格组成的串，其长度为串中空格字符的个数。

子串：串中任意个连续的字符组成的子序列。

主串：包含子串的串相应地称为主串。

字符在串中的位置：字符在序列中的序号。

串相等：当且仅当这两个串的值相等，即两个串的长度相等，并且各个对应位置的字符都相等。

通常，在程序中使用的串可分为两类：串变量和串常量。串变量和其他类型的变量一样，其取值是可改变的。串常量和常数一样，在程序中只能被引用但不能改变其值，即只能读不能写。

**【例 4-1】** 假设  $a$ 、 $b$ 、 $c$ 、 $d$  为如下的 4 个串： $a = 'Guang'$ ， $b = 'Zhou'$ ， $c = 'GuangZhou'$ ， $d = 'Guang Zhou'$ 。它们的长度分别为 5、4、9、10；并且  $a$  和  $b$  都是  $c$  和  $d$  的子串， $a$  在  $c$  和  $d$  中的位置都是 1，而  $b$  在  $c$  中的位置是 6，在  $d$  中的位置是 7； $a$ 、 $b$ 、 $c$ 、 $d$  彼此都不相等。

显然，若某串的长度为  $n$ ，则在该串中，长度为 1 的子串的个数为  $n$ ，长度为 2 的子串的个数为  $n-1$ ，……，长度为  $i$  的子串的个数为  $n-(i-1)$ ，所以长度为  $n$  的串中子串总数（包括空串与自身）为  $1 + \sum_{i=1}^n (n-i+1) = 1 + \frac{n(n+1)}{2}$ 。串的抽象数据类型定义见 ADT 4-1。

---

ADT String{

数据集合:  $D=\{ a_i \mid a_i/\text{CharacterSet}, i=1, 2, \dots, n, n\geq 0\}$

数据关系:  $R=\{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i/D, i=2, 3, \dots, n \}$

数据操作:

Assign\_Str(&T, chars) //生成串

输入: 字符串常量 chars。输出: 生成一个值等于 chars 的串 T。

Copy\_Str(&T, S) //复制串

输入: 串 S。输出: 串 S 复制为串 T。

Empty\_Str(S) //判断串是否为空

输入: 串 S。输出: 若 S 为空串, 则返回 TRUE; 否则, 返回 FALSE。

Compare\_Str(S, T) //比较两个串

输入: 串 S 和 T。

输出: 若  $S>T$ , 则返回值大于 0; 若  $S=T$ , 则返回值等于 0; 若  $S<T$ , 则返回值小于 0。

Length\_Str(S) //求串长度

输入: 串 S。输出: S 的元素个数, 称为串的长度。

Clear\_Str(&S) //清空串;

输入: 串 S。输出: 将 S 清为空串。

Concat\_Str(&T, S1, S2) //串的连接

输入: 串 S1 和 S2。输出: 用 T 返回由 S1 和 S2 连接而成的新串。

Sub\_String(&Sub, S, pos, len) //求子串

输入: 串 S,  $1\leq pos\leq \text{StrLength}(S)$  且  $0\leq len\leq \text{StrLength}(S) - pos + 1$ 。

输出: 用 Sub 返回串 S 的自第 pos 个字符起长度为 len 的子串。

Index(S, T, pos) //返回子串位置

输入: 串 S 和 T, T 是非空串,  $1\leq pos\leq \text{StrLength}(S)$ 。

输出: 若主串 S 中存在和串 T 值相同的子串, 则返回它在主串 S 中第 pos 个字符之后第一次出现的位置; 否则, 函数值为 0。

Replace(&S, T, V) //替换子串

输入: 串 S, T 和 V, T 是非空串。

输出: 用 V 替换主串 S 中出现的所有与 T 相等的非重叠的子串。

Insert\_Str(&S, pos, T) //在串的某个位置插入另一个串

输入: 串 S 和 T,  $1\leq pos\leq \text{StrLength}(S)+1$ 。

输出: 在串 S 的第 pos 个字符之前插入串 T。

Delete\_Str(&S, pos, len) //删除子串

输入: 串 S,  $1\leq pos\leq \text{StrLength}(S) - len + 1$ 。

输出: 从串 S 中删除从第 pos 个字符起长度为 len 的子串。

Destroy\_Str(&S) //撤销串

输入: 串 S。输出: 删除串 S。

}ADT String

---

### 4.1.2 串的大小比较

在实际应用中，常常需要比较串之间的大小关系。那么串之间的大小关系是如何定义的呢？

#### 1. 字符的大小

在计算机中，每个字符都有一个唯一的数值表示——字符编码（字符内码），字符间的大小关系就定义为它们的代码之间的大小关系。字符编码可有多种，对英文字母和符号，ASCII 码是最常见的一种。本书用字符的 ASCII 码之间的大小关系代表相应字符的大小关系。例如，字符 A 与 B 的 ASCII 码分别为 65 与 66（十进制数），则 ‘A’ < ‘B’。

ASCII 码即美国信息交换标准编码，是长度为 8 位二进制符号，所以最多能编  $2^8=256$  个不同的字符。但 ASCII 码中规定，最高位为 1 的码代表一些特殊的字符（或命令），所以 ASCII 码有效的字符编码为 128 个，其包括英文大、小写字母，数字 0~9 及一些常用符号（计算机键盘上的字符键）。在字符和数字中，空格编码最小，其次是数字（按 0~9 的次序）、大写字母（按 A~Z 的次序）、小写字母（按 a~z 的次序）等。

对于汉字，它们的大小关系也按编码大小确定。汉字的编码也有几种，如我国内地用的国标码（GB2312），我国台湾地区用的 Big5 等。GB2312（国标码）是针对汉字的 16 位二进制编码，所以最多能编  $2^{16}=65536$  个不同的码，足以代表新华字典中所有的汉字。GB2312 将汉字分为两级编码，分别称一级字库和二级字库。一级字库包括了日常用的大多数汉字，其汉字的国标码之间的大小关系，与对应的汉语拼音构成的串的大小关系一致（在新华字典中，排在较前面的字，它的国标码也较小）。而二级字库中的汉字码之间的大小关系，与对应汉字的笔画数目的大小关系一致。

要在一个系统中混合使用多种文字，需要一种统一的编码系统，它可以将各种文字的基本字符在同一空间内编码，Unicode 就是这样一种编码。

#### 2. 字符串间的大小关系

有了字符大小的定义，就可考虑串的大小关系了。设 X 与 Y 是两个串， $X = \langle x_1 x_2 \cdots x_m \rangle$ ， $Y = \langle y_1 y_2 \cdots y_n \rangle$ ，则它们之间的大小关系定义如下。

① 当  $m=n$  且  $x_i=y_i$ ， $\cdots$ ， $x_m=y_n$  时，称  $X=Y$ 。

② 当下列条件之一成立时，称  $X<Y$ ：

i.  $m<n$ ，且  $x_i=y_i$ ， $i=1, 2, \cdots, m$ ；

ii. 存在某一个  $k \leq \text{MIN}(m, n)$ ，使得  $x_i=y_i$ ， $i=1, 2, \cdots, k-1$ ， $x_k < y_k$ 。

③ 当不属于情况①与②时，称  $X>Y$ 。这种方法定义的串的大小关系，也称字典序。

下面给出几个例子说明此定义：‘abac’与‘abac’，相等(=)；‘we’与‘web’，前者小于后者（属于情况 i）；‘mouth’与‘move’，前者小于后者（属于情况 ii）

## 4.2 串的存储结构

与线性表类似，串也有两种基本的存储结构：顺序结构与链式结构。考虑到存储效率与算法实现的方便性，串多采用顺序存储结构。

## 1. 顺序存储

串的顺序存储结构简称为顺序串。与顺序表类似，顺序串用一组地址连续的存储单元来存储串中的字符序列，可用高级程序语言中字符数组来实现。按存储分配的不同可将顺序串分为静态存储分配的顺序串和动态存储分配的顺序串。

## 2. 链式存储

用单链表方式存储串的值，这种存储结构简称为链串，链串分为两种形式。①非压缩形式。一个链表结点只存储一个字符。为了操作方便，设置一个指向首元素结点的指针和一个指向尾元素结点的指针，同时仍设置串长度值字段。这种结构类似于线性表的链式存储，其优点是操作方便，但存储利用率很低。②压缩形式。一个链结点存储多个字符，实质上是一种连续存储与链式相结合的结构。这种结构能提高存储利用率，但其对应的操作实现起来较为复杂。例如，在对串长度进行修改时，常涉及链结点的增加与删除操作，如果仅允许不满的结点出现在最后一个结点上，则需要经常移动字符，但如果允许不满结点出现在其他位置，则又会造成串的表示不一致的问题，也会使操作实现复杂化。

# 4.3 串函数与串的类型定义

### 4.3.1 常用的C++串函数

C++的串库（string.h）中提供了许多字符串的操作函数，几个常用的 C++字符串函数及其使用方法如下。

假设已有以下定义语句：

```
char str1[50]="Happy birthday to", str2[ ]="coffeehu";
char temp1[100], temp2[6], *temp;
char str[ ]="This is a sentence with 7 tokens";
```

#### （1）串拷贝函数

char \*strcpy(char \*s1, const char \*s2)，将字符串 s2 复制到字符串数组 s1 中，返回 s1 的值。

char \*strncpy(char \*s1, const char \*s2, size\_tn)将字符串 s2 中最多 n 个字符复制到字符串数组 s1 中，返回 s1 的值。

例如，

```
strcpy(temp1, str1); strncpy(temp2, str1, 5);
temp2[5]='\0'; cout<<"strncpy result:"<<temp2<<"\n"; //输出 strncpy result:Happy
```

#### （2）串连接函数

char \*strcat(char \*s1, const char \*s2)，将字符串 s2 添加到字符串 s1 的后面，s2 的第一个字符重定义 s1 的 null 终止符，返回 s1 的值。

char \*strncat(char \*s1, const char \*s2, size\_tn)将字符串 s2 中最多 n 个字符添加到字符串

s1 的后面，s2 的第一个字符重定义 s1 的 null 终止符，返回 s1 的值。

例如，

```
cout<<"strcat result: "<<strcat(str1, str2)<<"\n";
//输出 strcat result:Happy birthday to coffeehu
cout<<"strncat result: "<<strncat(str1, str2, 6)<<"\n";
//输出 strncat result:Happy birthday to coffeehucoffee
```

(3) 串比较函数

int strcmp(const char \*s1, const char \*s2)，比较字符串 s1 和字符串 s2。函数在 s1 等于、小于或大于 s2 时，分别返回 0、小于 0 或者大于 0 的值。

int strncmp(const char \*s1, const char \*s2, size\_tn)比较字符串 s1 中的 n 个字符和字符串 s2。函数在 s1 等于、小于或大于 s2 时，分别返回 0、小于 0 或者大于 0 的值。

例如，

```
cout<<"strcmp result: "<<strcmp(temp2, "Happ")<<"\n"; //输出 strcmp result:1
cout<<"strncmp result: "<<strncmp(str1, "Happy", 5)<<"\n"; //输出 strncmp result:0
```

(4) 串长度函数

size\_strlen(const char \*s)，确定字符串长度，返回 null 终止符之前的字符数。例如，

```
cout<<"strlen result: "<<strlen(str2)<<"\n"; //输出 strlen result:8
```

(5) 串中字符定位

```
char *strchr(char *str, char ch); temp=strchr(str1,'b');
cout<<"strchr result: "<<temp<<endl; //输出 strchr result:birthday to coffeehucoffee
```

4.3.2 串的定义

C++语言提供的丰富的串函数库有很强的串操作功能，但对于一般使用者来说，其使用起来不是很方便。在一些高级程序设计语言中，两个串可用“+”连接，可用赋值号把一个串赋值给另一个串，两个串的比较可直接用“<”、“>”、“==”等比较运算符。这里可使用重载操作符的办法，使 C++语言中的一些运算符具有新的功能。程序 4-1 给出串的定义。

程序 4-1 串类的定义

```
class Str {
    unsigned char *room; //存储串的字符数组
    long size; //串的长度为 size-1
    long length; //串当前的长度
public:
    Str(long mSize=50); //构造函数
    Str(Str &s); //复制构造函数
    virtual ~Str();
    //...
```

```

//此处省略串的基本操作定义，详见 4.1.1 节
Str operator=(Str &ob); //赋值串对象
Str operator=(char *s); //赋值 C++ 串
Str operator +(Str &ob); //两个串对象的连接
Str operator +(char *s); //串对象与 C++ 的连接
friend Str operator+(char *s, Str &ob); //C++ 串与串对象的连接
//串对象之间的关系运算符
int operator==(Str &ob) { return !strcmp(room, ob.room); };
int operator!=(Str &ob) { return strcmp(room, ob.room); };
int operator<(Str &ob) { return strcmp(room, ob.room)<0; };
int operator>(Str &ob) { return strcmp(room, ob.room)>0; };
//类似重载<=,>=等运算符
//...
//串对象与 C++ 串之间的关系运算符
int operator==(char *s) { return !strcmp(room, s); };
int operator!=(char *s) { return strcmp(room, s); };
//类似重载<,>,<=,>=等运算符
//...
}

```

类 Str 的完整定义与实现，留作练习。

#### 【例 4-2】 求子串算法 substr。

```

void substr(char *sub, char a[ ], int i, int j)
{
    int k; int l=sizeof(a)-1;
    if (i>l) cout<< "此子串不存在。" <<endl;
    else {if ((i+j-1)<l) k=i+j-1; else k=l;
          for (p=i-1; p<k; p++) sub[p-i+1]=a[p];
        }
}

```

#### 【例 4-3】 串以链式形式存放的算法。

```

struct node
{
    node *next;
    char data;
};

void substr(node *sub, node *s, int i, int j)
{
    node *q=s->next; int l, k=1;
    while ((q!=NULL)&&(k<i)) {q=q->next; k++; }
    if (q==NULL) cout<<"子串不存在!"<<endl;
    else
    {
        node *p=new node; p->data=q->data; p->next=NULL;

```

```

sub=p; node *t=p; l=1; q=q->next;
while ((q!=NULL)&&(l<j))
{
    p=new node; p->data=q->data; p->next=t->next;
    t->next=p; t=p; l++; q=q->next; }
if (q==NULL) while (l<=j)
{
    p=new node; p->data=' '; p->next=t->next;
    t->next=p; t=p; l++; }
}

```

**【例 4-4】** 将 node 串 q 所指结点中第 p 个字符后的第 i 个有效字符送至 r。q 指针如图 4.1 所示。

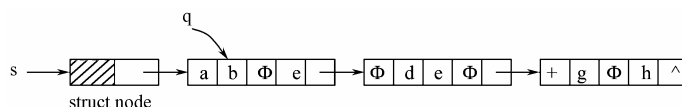


图 4.1 例 4-4 的图

```

struct node
{
    node *next;
    char *data;
};
char next(node *q, int p, int i)
{
    int j=i; p++;
    while (j>0)
    {
        if (q!=NULL)
        {
            if (p>4) {q=q->next; p=1; }
            else if (q->data[p]==' ') p++; //空白符
            else {p++; j--; }
        }
        else
        {
            cout<<"不存在第 i 个有效字符。"<<endl;
            j=-1; //准备 while 结束条件
        }
    }
    if (j==0) r=q->data[p-1];
}

```

## 4.4 串模式匹配

这里介绍一个很重要的操作——StrStr 的实现，该操作与 Index(S,T,pos)一样，也叫串模式匹配。给定两个串 T 与 P：T= ‘t<sub>0</sub>t<sub>1</sub>…t<sub>n-1</sub>’，P= ‘p<sub>0</sub>p<sub>1</sub>…p<sub>m-1</sub>’，其中 0≤m≤n，将在 T 中寻找最先出现的一个与 P 相同的子串的过程称为串模式匹配，称 T 为目标串（主串），P 称



为模式串（子串），下面介绍的 `StrStr()` 函数就属于串模式匹配。

串模式匹配在符号处理中占有十分重要的地位，它是基于规则匹配的逻辑推理的（如 Prolog 语言的目标执行过程、专家系统中的推理机），所以它的效率十分重要。这里先介绍一种简单的匹配算法，它的时间复杂度较高，然后介绍它的一种改进算法。

### 4.4.1 简单串模式匹配算法

这是一种带回溯的匹配算法（见程序 4-2）。首先将子串 `P` 视为从主串 `T` 中第 1 个字符起与 `T` 对齐，从头起依次比较对应字符；若全部对应相等，则表明已匹配成功，终止；否则，将 `P` 视为从 `T` 中第 2 个字符起与 `T` 对齐，再从头比较对应字符；过程与前类似，如此进行，直至某次匹配成功，或某次 `T` 中无足够的剩余字符与 `P` 对齐（不能匹配，失败）。

实现时，设置三个指示器 `i`，`j`，`ii`，它们的含义如下。

- `i`：指向主串 `T` 中当前参加比较的字符。起始时，指向 `T` 的首字符，之后，每比较一次，后移一步，一趟匹配失败时，回溯到该趟比较起点的下一位置。
- `j`：指向子串 `P` 中当前参加比较的字符。起始时，指向 `T` 的首字符，之后，每比较一次，后移一步，一趟匹配失败时，回溯到 `P` 的首字符处。
- `ii`：记录每趟比较在主串 `T` 中的起点，每趟比较后，后移一步。

程序 4-2 简单串模式匹配算法

```
long TStr::StrStr(TStr& s)
{
    //子串匹配。在对象串中寻找子串 s 的第 1 次出现的位置序号，未出现则返回-1
    long i,j,ii; i=j=ii=0;
    if (len<s.len) return -1; //若子串长度大于主串长度，则不能匹配
    do {
        if (room[i]==s.room[j]) {i++; j++;} //当前字符相等时推进
        else {ii++; i=ii; j=0;} //当前字符不等时回溯
        if (len-ii<s.len) return -1;
        //如果主串中尚未比较的字符个数小于子串长度，则不能匹配，失败返回
    } //else
    } while (j<s.len);
    return ii;
    //成功（子串中全部字符与主串中对应的字符相等，则匹配），返回子串位置
}
```

下面分析此算法的时间复杂度。显然，此算法的主要工作是 `do` 循环，它的循环次数与目标串及模式串相关，所以时间复杂度是个随机量。最理想的情况是，第一趟就得到匹配，此时的循环次数为  $n$ （即子串 `s` 的长度），而最坏的情况是不存在匹配，且每趟匹配过程都在比较完子串 `s` 中最后一个字符时才发现不能匹配，此种情况的循环次数是  $(m-n+1) \times n$ ，这里  $m$  为主串的长度。显然， $n$  越大，此式的值越小。当  $n \ll m$  时，此式约等于  $m \times n$ 。

### 4.4.2 无回溯的匹配算法

在上面介绍的匹配算法中，某趟匹配失败时，下一趟的匹配相当于将子串 `P` 后移 1 位再从头与主串中对应字符进行比较，即相当于 `i` 指示器回溯到上趟（最近失败的一趟）匹配

的起点的下一个位置，这样，主串中每个字符都要与子串中的第 1 个字符对应一次，再向后比较。因此，主串中每个字符参加比较的次数最多可达  $n$  次（ $n$  为子串长度），因此时间复杂度为  $O(nm)$ 。那么，能否使目标串中每个字符只参加一次比较呢？也就是说，能否不回溯  $i$  指示器？回答是肯定的。这个问题是由 D.E.Knoth 与 V.R.Pratt 和 J.H.Morris 同时解决的，所以有的文献也称这种思想的串匹配算法为 KMP 算法。

$i$  指示器之所以可不回溯，是因为重新开始一趟匹配时，可利用上趟匹配的结果。一般而言，一个无回溯匹配法的关键问题是，在匹配过程中一旦发现  $p_j$  与  $t_i$  不等，应能确定出从  $P$  中的哪个字符起与  $t_i$ （或  $t_{i+1}$ ）对齐继续往下比较。这里记这个字符为  $p_k$ ，显然  $k < j$ ，且对不同的  $i$ ， $k$  也不同。Knoth 等人发现这个  $k$  值仅仅依赖于子串  $P$  的前  $i$  个字符的构成，而与主串  $T$  无关。这是个令人鼓舞的发现，它是实现无回溯的关键。可用一个函数  $next$  表示  $j$  与  $k$  的这种对应关系，它是仅与子串有关的函数，其含义是，在匹配过程中，一旦发现一对不等的字符（设为  $p_j$  与  $t_i$ ），若  $next(j)=k$ （ $k \geq 0$ ），则下次的比较应从  $P$  中的  $p_k$  起与  $T$  中的  $t_i$  对齐往下进行；若  $next(j) < 0$ ，则  $P$  中任何字符不必再与  $t_i$  进行比较，下次比较从  $P$  的首字符起与  $t_{i+1}$  对齐往下进行。 $next(j)$  的取值，应首先保证使匹配过程无遗漏（即不放过任何可能的匹配），其次，应使  $P$  串向后滑动尽可能长的距离。函数  $next()$  称为失败函数。

下面假定对给定的子串，它的  $next$  函数已确立，则串模式匹配算法可描述为程序 4-3 的形式，它也是 KMP 算法的基本部分。

程序 4-3 简单串模式匹配算法

```
int StrStr(TStr T, TStr P)
{
    int i, j; int *next;
    if (P.len>T.len) return -1;
    next=BuildNext(P); //为 P 串构造失败函数对应的数组 next
    i=j=0; //假定串中首字符的下标为 0
    do {
        if (T.room[i]==P.room[j]) {i++; j++; }
        else {
            if (next[j]>0) j=next[j]-1;
            //下次，子串 P 的位置 j 与主串中当前位置 i 对齐比较
            else {j=0; i++; }
            //下次，子串的首字符与主串的当前比较位置的下个位置对齐比较
        }
        if (T.len-i<P.len-j) return -1; //不能继续匹配，失败返回
    } while (j<P.len);
    return i-P.len; //成功，返回出现位置
}
```

以上程序中，用一维数组  $next[]$  表示失败函数  $next()$ ，变量  $i$  只增不减，其初值为 0，在循环中一直保持  $i < n$ ，所以  $i$  加 1 的语句  $i++$  至多执行  $n$  次，又因为  $next[j] < j$ ，且  $1 \leq j \leq m$ ，故语句  $j=next[j]$  至多执行  $m$  次，由于循环中只存在分别含有  $i++$  与  $j=next[j]$  的两个互斥分支，故循环次数不超过  $MAX(m, n)$ ，因此算法的时间复杂度为  $O(MAX(m, n))$ 。通常  $m < n$ ，故时间复杂度为  $O(n)$ 。关于失败函数的构造，这里不作讨论。

# 4.5 串的应用——文本编辑

在办公室自动化、企业的现代化管理等许多领域中，文本编辑（程序）正发挥着越来越大的作用。文本编辑的实质是修改字符数据的形式或格式，如 Microsoft Word，汉字信息处理系统（激光排版系统）等。虽然这些系统的功能不太一样，但基本操作一般都包括串的查找、插入和删除等操作，主要用于源程序的输入和修改，报刊和书籍的编辑排版及公文书信的起草与润色等。通常，整个文本可看成是一个字符串，其中，页是文本的子串，行又是页的子串。例如，以下程序便可看成是一个文本串：

```
main() {
    int  x, y, z;
    scanf("%d, %d", &x, &y);
    z=x*y-10;
    printf(%d, z);
};
```

在该程序的输入过程中，由文本编辑程序为文本串建立相应的页表和行表，即建立各子串的存储映射。页表的每一项给出了页号和该页的起始行号，而行表的每一项则指示每一行的行号、起始地址和该行子串的长度。每输入的一行都可作为一个新的字符串加到文本中，串的值放在文本区中，行号、串值的存储地址及串长度则登记到表中。通过这个行表，新的一行可放到文本工作区的任何一个自由区中。设图 4.2 所示（注：↵为回车键）的文本串只占一页，且起始行号为 200，则该文本串的行表如图 4.3 所示。

m	a	i	n	(	)	{	↵			i	n	t		x	,	y	,	z	;
↵	s	c	a	n	(	"	%	d	,	%	d	"	,	&	x	,	&	y	)
;	↵	z	=	x	*	y	-	10	;	↵	p	r	i	n	t	f	(	%	d
,	z	)	;	↵	}	↵													

图 4.2 文本格式示例

行号	起始地址	长度
200	301	11
201	312	9
202	321	7
203	328	21
204	349	13
205	362	6

图 4.3 图 4.2 所示文本串的行表

下面讨论 3 种文本编辑的操作。

- （1）插入。把新插入的行存到文本的末尾，并按行号的次序把新行的行号、起始位置及该行字符串的长度信息插入到行表的合适位置。
- （2）删除。把被删除行的行号、起始位置及该行字符串的长度信息从行表中删去（若

存储空间较紧张，则应同时从内存中删去该行)。若被删除的行是所在页的起始行，则还要修改页表中相应页的起始行号(修改为下一行的行号)。

(3) 修改。①若新行长小于等于原行长，则把新行字符串仍存于原行字符串的位置，并修改行表中该行子串的长度信息即可；②若新行长大于等于原行长，则把新行字符串存于文本末尾，即为该行重新分配存储空间(并可删去文本中的原行)，并修改行表中该行的起始位置及行长信息。

# 本章总结

## 1. 学习要点

本章主要介绍串的基本概念、基本运算、静态存储结构与动态存储结构，以及基本操作的实现方法，串操作的综合应用程序设计。主要学习要点如下：① 串的逻辑结构定义和串的连接、求子串、定位、置换、插入、删除等基本运算的功能；② 串的静态存储和动态存储的基本思想和实现方法；③ 串操作在文本编辑、文本格式化处理中的应用。

## 2. 基本要求

- (1) 深入领会串运算的定义和基本算法以及综合应用程序设计；
- (2) 知道(字符)串的有关术语、逻辑结构、特点以及与线性表的区别；
- (3) 弄清串的连接，求子串、模式匹配、置换等基本运算的功能，并能灵活应用串的这些基本运算；
- (4) 弄清顺序串和链串紧缩格式的主要优缺点，非紧缩格式的优缺点；
- (5) 了解串及其运算在文本编辑及文本格式化中的应用。

## 3. 重点与难点

重点是：求子串、定位、置换基本运算(操作)的实现算法。难点是：串的综合应用程序设计。

# 习题 4

- 4-1 简述下列每对术语的区别：空串与空格串，串变量与串常量，主串与子串，串变量的名字与串变量的值。
- 4-2 在顺序串上实现串的判等运算  $\text{equal}(s, t)$  的算法，在链串上实现判等运算  $\text{equal}(s, t)$  的算法。
- 4-3 已知  $s = \text{'(xyz)+*'} , t = \text{'(x+z)*y'}$ ，试利用连接、求子串和置换等基本运算，将  $s$  转化为  $t$ 。
- 4-4 对于给定字符串， $1234+ \rightarrow /15678$ ，试写出删除串中的  $\text{' + \rightarrow /'}$  的算法。
- 4-5 试写出一个将已知字符串所有字符倒过来重新排列的算法。例如， $ABCDEF$  改为  $FEDCBA$ 。
- 4-6 设  $x$  和  $y$  为两个字符串，求在  $x$  中第一次出现的，而在  $y$  中不出现的字符位置，写出算法。
- 4-7 写出一个将串  $s_1$  插入到串  $s_2$  的某个字符之后的算法。例如， $s_1 = \text{'ABCDEF'}$ ， $s_2 = \text{'XYZ'}$ ，将  $s_2$  插入的  $s_1$  的  $B$  之后，假定串值都以链表方式存储，其结点大小为 4，链域大小为 2。
- 4-8 设串以向前链表结构存储，试给出将串中元素倒过来重新排列的算法。

4-9 试给出等量分块存储结构下等量分块插入运算的算法，要求：①分析插入时的各种情况及处理对策；②设块大小为 4，写出其算法思想；③按所述思想给出算法。

4-10 分别将顺序结构和链式结构下的求子串 `substr` 运算用 C++ 语言实现。

4-11 设计一个算法，删去串  $S$  中从第  $i$  个字符开始的  $j$  个字符，说明算法所用的存储结构，并估计算法的执行时间。

4-12 若  $x$  和  $y$  是两个单链表存储的串，设计一个算法，找出  $x$  中第一个不在  $y$  中出现的字符。

4-13 若  $s$  和  $t$  是用结点大小为 1 的单链表存储的两个串，设计一个算法，将串  $s$  中首次与串  $t$  匹配的子串逆置。

4-14 假设所使用的串采用顺序存储结构，试编写一个实现 `insert(s, s1, i)` 运算的程序和 `delete(s, i, j)` 运算的程序。改为采用链式存储结构（每个结点存放一个字符），重复上述两个问题。

4-15 假设所使用的串采用链接存储结构，链中每个结点存放  $m$  ( $m=4$ ) 个字符，试编写一个实现 `insert(s, s1, i)` 运算的程序过程。为了减少移动字符的次数，允许使用无效的特殊字符，但一个结点至少含有一个有效字符。另编写 `delete(s1, i, j)` 运算的程序过程。

4-16 设串采用链表结构存储，利用串的基本运算，给出下列算法：①把一个串中从左到右最先出现的 `sb$` 改变成 `c$`；②把一个串中从左到右第三次出现的 `sb$` 改变成 `c$`；③把一个串中从左到右出现的全部 `sb$` 改变成 `c$`；④把一个串中从左到右第三次及以后出现的 `sb$` 改变成 `c$`；⑤把一个串中倒数第一、第二、第三个出现的 `sb$` 改变成 `c$`（提示：利用一个栈，找出倒数第一、第二、第三个 `sb$` 的位置）。

4-17 设某校学生学号含义如下： $xx$ （年级） $xx$ （系） $xx$ （专业班） $xx$ （学生号）。又知，该校在使用计算机管理时，一个系某专业的档案全部存放在一起，一个班的成绩放在一起，档案文件名的第一个字符为“D”，成绩文件名的第一个字符为“F”，试用串的运算形成这两个文件的文件名。

## 第5章 多维数组与广义表

多维数组与广义表均可看成是线性表的一种扩展，即表中的数据元素本身也是一个数据结构，元素的值是可再分解的。它们的逻辑特征是，一个数据元素可能有多个直接前驱和多个直接后继。尽管 C++ 支持多维数组，但它无法保证数组下标的合法性。同时，C++ 也未能提供数组的输入、输出以及简单的算术运算（如数组赋值和数组相加）。为了克服这些不足，本章针对一维数组设计了类 Array1D。

### 5.1 数 组

#### 5.1.1 数组的定义

与线性表一样，数组中所有的数据元素都必须属于同一数据类型，其特点是，每个数据元素可以又是一个线性表结构。若线性表中的数据元素为简单元素，则称为一维数组，即向量；若一维数组中的数据元素又是一维数组，则称为二维数组；依次类推，若二维数组中的元素又是一个一维数组结构，则称为三维数组。因此，线性表结构可看成是数组结构的一个特例，而数组结构则是线性表结构的扩展。

例如，在图 5.1 (a) 所示的  $m$  行  $n$  列二维数组中，每个数据元素  $a_{ij}$  都受两个关系——行关系和列关系的约束。在行关系中， $a_{i,j+1}$  是  $a_{ij}$  的直接后继元素。在列关系中， $a_{i+1,j}$  是  $a_{ij}$  的直接后继元素，该数组元素的个数为  $m \times n$ 。该数组可被看成是一个线性表  $A=(a_0, a_1, \dots, a_p)$ ， $p = m-1$  或  $n-1$ ，其中每个元素  $a_j$  可以是如图 5.1 (b) 所示的列向量形式线性表  $a_j=(a_{0,j}, a_{1,j}, \dots, a_{m-1,j})$ ， $0 \leq j \leq n-1$ ，也可以是如图 5.1 (c) 所示的行向量形式线性表  $a_i=(a_{i,0}, a_{i,1}, \dots, a_{i,n-1})$ ， $0 \leq i \leq m-1$ 。

$$\begin{aligned}
 A_{m \times n} &= \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{pmatrix} & A_{m \times n} &= \left( \begin{pmatrix} a_{0,0} \\ a_{1,0} \\ \cdots \\ a_{m-1,0} \end{pmatrix} \begin{pmatrix} a_{0,1} \\ a_{1,1} \\ \cdots \\ a_{m-1,1} \end{pmatrix} \cdots \begin{pmatrix} a_{0,n-1} \\ a_{1,n-1} \\ \cdots \\ a_{m-1,n-1} \end{pmatrix} \right) \\
 &\text{(a) 矩阵形式表示} & & \text{(b) 列向量的一维数组} \\
 A_{m \times n} &= ((a_{0,0} a_{0,1} \cdots a_{0,n-1}), (a_{1,0} a_{1,1} \cdots a_{1,n-1}), \cdots, (a_{m-1,0} a_{m-1,1} \cdots a_{m-1,n-1})) \\
 & & & \text{(c) 行向量的一维数组}
 \end{aligned}$$

图 5.1 二维数组图例

同理， $n$  维数组中每个数据元素都受  $n$  个关系的约束，都对应于一组下标  $(j_1, j_2, \dots, j_n)$ ，元素  $a_{j_1, j_2, \dots, j_n}$  ( $1 \leq j_i \leq b_i-1$ ,  $i = 1, 2, \dots, n$ ) 有一个直接后继元素。因此，就单个关系而言，仍是线性关系。类似于线性表，数组的抽象数据类型定义参见 ADT 5-1。

#### ADT 5-1 数组的抽象数据类型描述

ADT Array {

数据集合:  $j_i=0, \dots, b_i-1$ ,  $i=1, 2, \dots, n$

$D = \{a_{j_1 j_2 \dots j_n} \mid n (>0) \text{ 称为数组的维数, } b_i \text{ 是数组第 } i \text{ 维的长度,}$

$j_i \text{ 是数组元素的第 } i \text{ 维下标, } a_{j_1 j_2 \dots j_n} \in \text{ElemSet}\}$

数据关系:  $R = \{R_1, R_2, \dots, R_n\}$

$R_i = \{ \langle a_{j_1 \dots j_i \dots j_n}, a_{j_1 \dots j_i+1 \dots j_n} \rangle \mid 0 \leq j_k \leq b_i-1, 1 \leq k \leq n \text{ 且 } k \neq i,$

$0 \leq j_i \leq b_i-2, a_{j_1 \dots j_i \dots j_n}, a_{j_1 \dots j_i+1 \dots j_n} \in D, i=2, \dots, n\}$

数据操作:

Init\_Array(&A, n, bound<sub>1</sub>, ..., bound<sub>n</sub>) //初始化多维数组

输出: 若维数  $n$  和各维长度合法, 则构造相应的数组 A, 并返回 TRUE。

Destroy\_Array(&A) //撤销多维数组

输出: 销毁数组 A。

Value\_Array(A, &e, index<sub>1</sub>, ..., index<sub>n</sub>) //取多维数组的某个元素值

输入:  $n$  维数 A 组, e 为元素变量, 随后是  $n$  个下标值。

输出: 若各下标不超界, 则 e 赋值为所指定的 A 的元素值, 并返回 TURE。

Assign\_Array(&A, e, index<sub>1</sub>, ..., index<sub>n</sub>) //给多维数组的元素赋值

输入:  $n$  维数 A 组, e 为元素变量, 随后是  $n$  个下标值。

输出: 若下标不超界, 则将 e 的值赋给所指定的 A 元素值, 返回 TRUE。

}ADT Array

### 5.1.2 C++的数组

尽管在 C++中数组是一个标准的数据结构, 但 C++数组的索引 (也称为下标) 必须采用  $[i_1][i_2][i_3] \dots [i_k]$  的形式, 其中,  $i_k$  为非负整数。如果  $k$  为 1, 则数组为一维数组; 如果  $k$  为 2, 则数组为二维数组。 $i_1$  是索引的第一个坐标,  $i_2$  是第二个坐标,  $i_k$  是第  $k$  个坐标。在 C++中, 值为整数类型的  $k$  维数组 score 可用 `int score[u1][u2][u3]...[uk]` 语句来创建, 其中,  $u_i$  是正的常量或由常量表示的正的表达式。对于这样一个数组描述, 索引  $i_j$  的取值范围为:  $0 \leq i_j < u_j, 1 \leq j \leq k$ 。因此, 该数组最多可容纳  $n = u_1 u_2 u_3 \dots u_k$  个值。由于数组 score 中的每个值都是整数, 所以需要占用 `sizeof(int)` 个字节, 因此, 整个数组所需要的内存空间为 `sizeof(score) = n * sizeof(int)` 个字节。C++编译器将为数组预留这么多空间。如果预留空间的起始地址为 start, 则该空间将延伸至 `start + size(score) - 1`。

### 5.1.3 数组的存储结构与寻址问题

数组是一种特殊的数据结构, 一般要求元素的存储地址能根据它的下标 (即逻辑关系) 计算出来, 因此数组一般只采用顺序存储结构。讨论数组元素地址时, 一般将第 1 个元素的起始存储单元作为参考单元, 参考单元的绝对地址称为该数组的首地址, 数组其他元素的相对地址均相对于首元素的起始地址。设  $i_1, i_2, \dots, i_n$  为某  $n$  维数组中的一个元素的下标, 则用  $\text{Loc}(a_{i_1, i_2, \dots, i_n})$  表示此元素的相对地址。

对一维数组, 与线性表类似, 可使用顺序存储方式; 但对多维数组, 由于其已不属于线性结构的范畴, 因此不能直接使用顺序存储方式。但多维数组有其特殊性, 具有线性结构的痕迹, 它可唯一地转换为一维结构, 并可根据元素的存储位置推算出元素的逻辑关系 (下标), 所以可将多维数组映射为一维结构, 然后使用顺序存储方式。

1. 一维数组

一维数组的每个元素只含一个下标，其实质上是一个线性表，存储方法与普通线性表的顺序存储方法完全相同，即将数组元素按它们的逻辑次序存储在一片连续区域内。设一维数组为  $A=(a_0, a_1, \dots, a_{n-1})$ ，则它的元素  $a_i$  的相对地址为  $Loc(a_i) = i \times c$ 。这里， $c$  表示每个元素占用的存储单元数目。

2. 二维数组

二维数组的每个元素含有两个下标，不是一般的线性表。如果将二维数组的第 1 个下标理解为行号，第 2 个下标理解为列号，然后按行列次序排列各元素，则二维数组呈阵列形式。对于这种非线性结构的存储，需将多维关系映射为一维（线性）关系，即要确定多维到一维的映射。以图 5.1（a）所示的二维数组为例，它有两种存储方式：以列序为主序的存储方式[见图 5.2（a）]和以行序为主序的存储方式[见图 5.2（b）]。

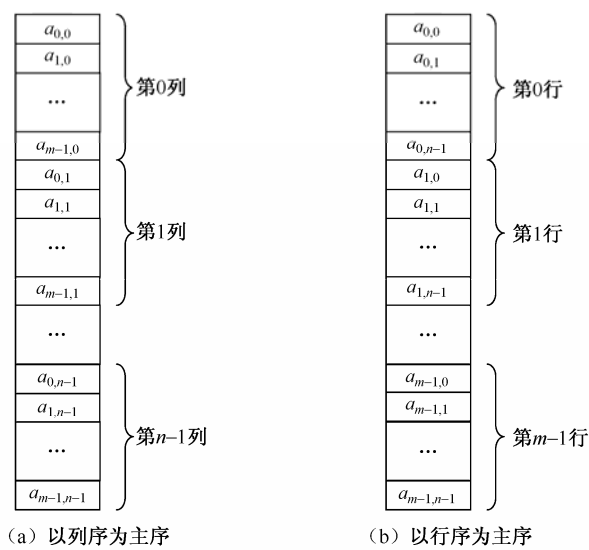


图 5.2 二维数组的两种存储方式

(1) 按行存储

按行存储的基本思想是先行后列，即先存储行号较小的元素，对于行号相同者，先存储列号较小者。设二维数组的行下标与列下标变化范围分别为闭区间  $[l_1, h_1]$  与  $[l_2, h_2]$ ，按行存储，则它的任一元素  $a_{i,j}$  的相对地址计算公式为  $Loc(a_{i,j}) = ((h_2-l_2+1)(i-l_1)+(j-l_2)) \times c$ ，这里  $c$  为每个元素所占单元数目， $i \in [l_1, h_1]$ ， $j \in [l_2, h_2]$  且  $i$  与  $j$  为整数。

**【例 5-1】** 设某二维数组的两个下标的范围分别为  $[-1, 2]$  与  $[0, 1]$ ，则它的元素按行存储的次序为（用  $a_{i,j}$  表示元素，每个元素占两个单元）：

相对地址	0	2	4	6	8	10	12	14
元素	$a_{-1,0}$	$a_{-1,1}$	$a_{0,0}$	$a_{0,1}$	$a_{1,0}$	$a_{1,1}$	$a_{2,0}$	$a_{2,1}$

它的元素  $a_{-1,1}$  的相对地址计算如下  $Loc(a_{-1,1})=((1-0+1)(-1+1)+(1-0)) \times 2=2$ 。



## (2) 按列存储

按列存储的基本思想是先列后行，即先存储列号较小者，对于列号相同者，先存储行号较小者。设二维数组的行下标与列下标变化范围分别为闭区间 $[l_1, h_1]$ 与 $[l_2, h_2]$ ，按列存储，每个元素占  $c$  个存储单元，则它的任一元素  $a_{i,j}$  的相对地址计算公式为  $\text{Loc}(a_{i,j})=((j-l_2)(h_1-l_1+1)+(i-l_1))\times c$ 。对二维数组，还可有其他的二维向一维的映射方法，如按正对角线、反对角线等，这里不作讨论。

## 3. 多维数组

下面将二维数组推广到多维数组，对  $n$  维数组也采用两种存储映射方式，即二维数组的按行存储与按列存储的推广。按行存储和按列存储都是一种“左”下标优先的存储方法，即第 1 个（最左）下标的下标值较小的元素较先存储，对于第 1 个下标值相同者，按第 2 个下标优先存储，对任意的  $k>1$ ，对第  $1\sim(k-1)$  维相同者，先存储第  $k$  维中下标值较小者。

**【例 5-2】** 设某三维数组  $a$  的三个下标范围分别为 $[2, 4]$ ， $[0, 1]$ ， $[1, 3]$ ，则它按行存储的次序为

$a_{2,0,1}$   $a_{2,0,2}$   $a_{2,0,3}$   $a_{2,1,1}$   $a_{2,1,2}$   $a_{2,1,3}$   $a_{3,0,1}$   $a_{3,0,2}$   $a_{3,0,3}$   $a_{3,1,1}$   $a_{3,1,2}$   $a_{3,1,3}$   
 $a_{4,0,1}$   $a_{4,0,2}$   $a_{4,0,3}$   $a_{4,1,1}$   $a_{4,1,2}$   $a_{4,1,3}$

设  $n$  维数组第  $k$  维的下标范围是 $[l_k, h_k]$ ，记  $d_k=h_k-l_k+1$ ， $j_k=i_k-l_k$ ， $k=1, 2, \dots, n$ 。显然， $d_k$  为第  $k$  维的体积（元素个数）。设每个元素占  $c$  个单元，则下标为  $i_1, i_2, \dots, i_n$  的元素的相对地址的计算公式为  $\text{Loc}(a_{i_1, i_2, \dots, i_n})=c\times(j_1d_2d_3\cdots d_n+j_2d_3\cdots d_n+\cdots+j_{n-1}d_n+j_n)$ 。例如，三维数组的寻址公式为  $\text{Loc}(a_{i_1, i_2, i_3})=c\times(j_1d_2d_3+j_2d_3+j_3)$ 。 $n$  维数组寻址公式可应用数学归纳法证得。

## 4. 寻址公式的计算

下面考虑如何根据一组给定下标求出对应数组元素地址的问题，这是数组最重要的基本操作，一般用在高级语言的实现当中。这里只考虑  $n$  维数组按行存储的寻址公式的计算。用秦九韶法变换按行存储公式中的主要部分：

$$\begin{aligned}& j_1d_2d_3\cdots d_n+j_2d_3\cdots d_n+\cdots+j_{n-1}d_n+j_n \\& =(j_1d_2+j_2)d_3\cdots d_n+j_3d_4\cdots d_n+\cdots+j_{n-1}d_n+j_n \\& =((j_1d_2+j_2)d_3+j_3)d_4\cdots d_n+j_4d_5\cdots d_n+\cdots+j_{n-1}d_n+j_n \\& =\cdots \\& =(\cdots(j_1d_2+j_2)d_3+j_3)d_4+j_4)d_5+\cdots+j_{n-1})d_n+j_n \\& =(\cdots((0\times d_1+j_1)d_2+j_2)d_3+j_3)d_4+j_4)d_5+\cdots+j_{n-1})d_n+j_n\end{aligned}$$

此式的值乘以元素占用的单元数  $c$  即为元素  $(i_1i_2\cdots i_n)$  的相对地址 ( $j_k=i_k-l_k$ )。此式可按下法计算：

$s=0; k=1; \text{ while } (k\leq n) \quad \{s=s*d_k+j_k; k=k+1;\}$

下面将该计算过程写成一个 C 函数，用 3 个一维数组  $l$ 、 $h$ 、 $i$  分别表示  $n$  个下标的下界、上界及待求地址的元素的  $n$  个下标值，见程序 5-1。

---

```

long GetArrayElemAddress(long l[ ], long h[ ], long i[ ], long n, int c)
{
    long k, s; s=0;
    for (k=0; k<n; k++) //注意, 这里是从 0 号元素开始存储内容的
        s=s*(h[k] - l[k]+1)+(i[k] - l[k]);
    return s;
}

```

---

按列存储公式的计算与按行存储的类似, 留作练习。

## 5.2 类Array1D

尽管 C++语言支持一维数组, 但这种支持很不够, 对于一些特殊操作会有很大的限制。例如, 使用超出正常范围之外的索引值来访问数组。考察如下的数组 `int a[9]`, 可访问数组元素 `a[-3]`, `a[9]`和 `a[90]`, 尽管-3, 9 和 90 是非法的索引。允许使用非法的索引通常会使用程序产生无法预料的行为并给调试带来困难。C++数组不能使用如下的语句来输出数组:

```
cout << a << endl;
```

不能对一维数组进行诸如加法和减法等运算。为了克服这些不足, 本节定义了类 `Array1D` (见程序 5-2), 该类的每个实例 `X` 都是一个一维数组。`X` 的元素存储在数组 `X.element` 之中, 第 `i` 个元素位于 `X.element[i]`中,  $0 \leq i < \text{size}$ 。

程序 5-2 一维数组的类定义

---

```

template<class T>
class Array1D {
public:
    Array1D(int size=0);
    Array1D(const Array1D<T>&v);    //复制构造函数
    ~Array1D( ) { delete[ ] element; }
    T& operator[ ](int i) const;
    int Size( ) {return size; }
    bool SetElem(int k, const T&x);    //设置第 k 个元素值
    bool GetElem(int k, T & x);    //获取第 k 个元素
    Array1D<T>& operator=(const Array1D<T>&v);
    Array1D<T> operator+( ) const;    //一元加法操作符
    Array1D<T> operator+(const Array1D<T>&v) const;
    Array1D<T> operator - ( ) const;    //一元减法操作符
    Array1D<T> operator - (const Array1D<T>&v) const;
    Array1D<T> operator*(const Array1D<T>&v) const;
    Array1D<T>& operator+=(const T& x);
private:
    int size;
    T *element;    //一维数组
};

```

---

这个类的共享成员包括构造函数、复制构造函数、析构函数、下标操作符[ ]、返回数组大小的函数 Size，以及算术操作符+、-、\*和+=。此外，还可添加其他的操作符。程序 5-3 给出了构造函数和复制构造函数的代码。构造函数违背了 ANSIC++的要求，它允许数组具有 0 个元素。如果不希望出现这种违规行为，可对这段代码进行适当的修改。

### 程序 5-3 一维数组的构造函数

```
#include
template<class T>
Array1D<T>::Array1D(int sz)
{ //一维数组的构造函数
    if (sz<0) throw BadInitializers();
    size=sz; element=new T[sz]; }
template<class T>
Array1D<T>::Array1D(const Array1D<T>&v)
{ //一维数组的复制构造函数
    size=v.size; element=new T[size]; //申请空间
    for (int i=0; i<size; i++) element[i]=v.element[i]; //复制元素
}
```

程序 5-4 给出了重载操作符[ ]的代码。该操作符用来返回指向第 i 个元素的引用，这样可使存储和查询操作按照很自然的方式进行。利用这个操作符，使用语句 X[1]= 2\*Y[3]，就可按照期望的方式进行工作，其中 X 和 Y 的类型均为 Array1D。代码 Y[3]在对象 Y 上施加操作符[ ]，结果返回一个指向元素 3 的引用，然后将它乘以 2。代码 X[1]也调用了操作符[ ]，结果返回一个指向 X[1]的引用，之后，2\*Y[3]的结果将存储在这个引用位置上。

### 程序 5-4 重载下标操作符 “[ ]”

```
template<class T>
T& Array1D<T>::operator[ ](int i) const
{ //返回指向第 i 个元素的引用
    if (i<0||i>=size) throw OutOfBounds(); else { return element[i]; }
}
```

程序 5-5 给出了赋值操作符的代码。这段代码通过验证等号左右两边是否相同来避免进行自我赋值（即 X=X）。为了进行赋值，应该首先释放目标数组\*this 所占用的空间，然后分配能够容纳源数组 v 的空间。尽管对 new 的调用可能会失败，但并没有捕获所引发的异常，而是把它留给更适合处理这种异常的代码来捕获。如果 new 没有引发异常，则把源数组中的元素逐个复制到目标数组中。

### 程序 5-5 重载赋值操作符 “=”

```
template<class T>
Array1D<T>& Array1D<T>::operator=(const Array1D<T>& v)
{ //重载赋值操作符 “=”
    if (this!=&v) { //不是自我赋值
        size=v.size; delete[ ] element; //释放原空间
```

```

        element=new T[size]; //申请空间
        for (int i=0; i<size; i++) element[i]=v.element[i]; //复制元素
    }
    return *this;
}

```

程序 5-6 给出二元减法操作符、一元减法操作符和增值操作符的代码，其他操作符的代码与此类似。

程序 5-6 重载二元减法操作符、一元减法操作符和增值操作符

```

template<class T>
Array1D<T> Array1D<T>::operator-(const Array1D<T>& v) const
{ //返回 w=(*this)-v
    if (size!=v.size) throw SizeMismatch( );
    Array1D<T> w(size); //创建结果数组 w
    for (int i=0; i<size; i++) w.element[i]=element[i]-v.element[i];
    return w;
}

template<class T>
Array1D<T> Array1D<T>::operator-( )const
{ //返回 w=-(*this)
    Array1D<T> w(size); //创建结果数组 w
    for (int i=0; i<size; i++) w.element[i]=-element[i]; return w;
}

template<class T>
Array1D<T>& Array1D<T>::operator+=(const T& x)
{ //把 x 加到(*this)的每个元素上
    for (int i=0; i<size; i++) element[i]+=x; return *this;
}

```

复杂度分析：当 T 是一个内部 C++数据类型（如 int, float 和 char）时，构造函数和析构函数的复杂度分析为  $O(1)$ ；而当 T 是一个用户自定义的类时，构造函数和析构函数的复杂度为  $O(\text{size})$ 。存在这种差别是因为，当 T 是一个用户自定义类时，在用 new(delete)创建（删除）数组 element 的过程中，对于 element 中的每个元素都要调用一次 T 的构造函数（析构函数）。下标操作符[]的复杂度为  $O(1)$ ，其他操作符的复杂度均为  $O(\text{size})$ （注意，复杂度不会是  $O(\text{size})$ ，因为所有操作符的代码都可引发一个异常并提前终止）。

## 5.3 矩阵的压缩存储

矩阵是很多科学与工程计算问题研究的数学对象。在高级语言程序设计中，常用二维数组来存储矩阵元素，有些程序语言还提供了各种方便用户使用的矩阵运算。然而，对于在数值分析中经常出现的一些阶数很高，且矩阵中有许多值相同的元素或者是零元素的特殊矩阵，不适合用二维数组来存放，因为这将造成大量存储空间的浪费。为了节省存储空间，可

对这类矩阵进行压缩存储，即为多个值相同的元素只分配一个存储空间，对零元素不分配空间。

### 5.3.1 特殊矩阵

所谓特殊矩阵是指非零元素或零元素的分布有一定规律的矩阵。常见的有对称矩阵、三角矩阵和对角矩阵等。

#### 1. 对称矩阵

##### (1) 定义

若  $n$  阶矩阵  $A$  中的元素满足下述性质  $a_{i,j}=a_{j,i}, 1\leq i, j\leq n$ ，则称  $A$  为  $n$  阶对称矩阵。图 5.3 所示为一个 4 阶对称矩阵。

##### (2) 对称矩阵的压缩存储

对称矩阵中的元素关于主对角线对称，故只要存储矩阵上三角或下三角中的元素，让每两个对称的元素共享一个存储空间，这样就能节约近一半的存储空间。

① 按“行优先顺序”存储主对角线（包括对角线）以下的元素（见图 5.4）

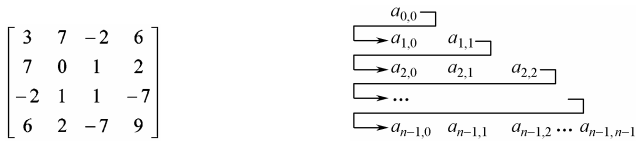


图 5.3 对称矩阵示例

图 5.4 对称矩阵的行优先顺序存放

即按  $a_{0,0}, a_{1,0}, a_{1,1}, \dots, a_{n-1,0}, a_{n-1,1}, \dots, a_{n-1,n-1}$  次序存放在一个向量  $sa[0 \dots n(n+1)/2-1]$  中（在下三角矩阵中，元素总数为  $\lceil n(n+1)/2 \rceil$ ），其中， $sa[0]=a_{0,0}, sa[1]=a_{1,0}, \dots, sa[n(n+1)/2-1]=a_{n-1,n-1}$ 。

##### ② 元素 $a_{i,j}$ 的存放位置

$a_{i,j}$  元素前有  $i$  行（从第 0 行到第  $i-1$  行），一共有  $1+2+\dots+i = i \times (i+1)/2$  个元素；在第  $i$  行上， $a_{i,j}$  之前有  $j$  个元素（即  $a_{i,0}, a_{i,1}, \dots, a_{i,j-1}$ ），因此有  $sa[i \times (i+1)/2 + j] = a_{i,j}$ 。

##### ③ $a_{i,j}$ 和 $sa[k]$ 之间的对应关系

若  $i \geq j$ ，则  $k = i \times (i+1)/2 + j, 0 \leq k < n(n+1)/2$ ；若  $i < j$ ，则  $k = j \times (j+1)/2 + i, 0 \leq k < n(n+1)/2$ ；令  $I = \max(i, j), J = \min(i, j)$ ，则  $k$  和  $i, j$  的对应关系可统一为  $k = I \times (I+1)/2 + J, 0 \leq k < n(n+1)/2$ 。

##### (3) 对称矩阵的地址计算公式

$$\text{Loc}(a_{i,j}) = \text{Loc}(sa[k]) = \text{Loc}(sa[0]) + k \times d = \text{Loc}(sa[0]) + [I \times (I+1)/2 + J] \times d$$

式中， $d$  为每个元素所占的内存单元个数。通过下标变换公式，能立即找到矩阵元素  $a_{i,j}$  在其压缩存储表示  $sa$  中的对应位置  $k$ ，因此是随机存取结构。

#### 2. 三角矩阵

##### (1) 三角矩阵的划分

以主对角线划分，三角矩阵有上三角矩阵和下三角矩阵两种。

① 上三角矩阵：矩阵的下三角（不包括主对角线）中的元素均为常数  $c$ 。

② 下三角矩阵：与上三角矩阵相反，矩阵的主对角线上方均为常数  $c$ 。

**注意：**在多数情况下，三角矩阵的常数  $c$  为零。

(2) 三角矩阵的压缩存储

三角矩阵中的重复元素  $c$  可共享一个存储空间，其余的元素正好有  $n \times (n+1)/2$  个，因此，三角矩阵可压缩存储到向量  $sa[0.. n(n+1)/2]$  中，其中， $c$  存放在向量的最后一个分量中。

① 上三角矩阵中  $a_{i,j}$  和  $sa[k]$  之间的对应关系

在上三角矩阵中，主对角线之上的第  $p$  行 ( $0 \leq p < n$ ) 恰有  $n-p$  个元素，按行优先顺序存放上三角矩阵中的元素  $a_{i,j}$  时， $a_{i,j}$  元素前有  $i$  行 (从第 0 行到第  $i-1$  行)，一共有  $(n-0)+(n-1)+(n-2)+ \cdots + (n-i) = i \times (2n-i+1)/2$  个元素；在第  $i$  行上， $a_{i,j}$  之前有  $j-i$  个元素 (即  $a_{i,j}, a_{i,j+1}, \cdots, a_{i,j-1}$ )，因此有  $sa[i \times (2n-i+1)/2 + j - i] = a_{i,j}$ 。所以

$$k = \begin{cases} i \times (2n-i+1)/2 + j - i & \text{当 } i \leq j \text{ 时} \\ n \times (n+1)/2 & \text{当 } i > j \text{ 时} \end{cases}$$

② 下三角矩阵中  $a_{i,j}$  和  $sa[k]$  之间的对应关系

$$k = \begin{cases} i \times (i+1)/2 + j & \text{当 } i \geq j \text{ 时} \\ n \times (n+1)/2 & \text{当 } i < j \text{ 时} \end{cases}$$

**注意：**三角矩阵的压缩存储结构是随机存取结构。

(3) 对角矩阵

在这种矩阵中，所有的非零元素都集中在以主对角线为中心的带状区域中。即除了主对角线上和直接在对角线上、下方若干条对角线上的元素之外，其他的元素均为零，如图 5.5 所示。对这种矩阵也可按某种原则 (或以行为主，或以对角线的顺序为主) 将其压缩存储到一维数组上。

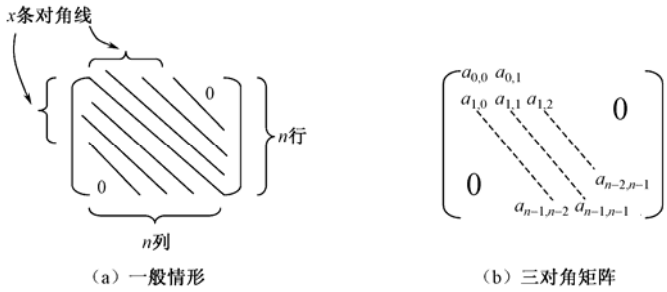


图 5.5 对角矩阵

总之，上述这些特殊矩阵中非零元素的分布都有一个明显的规律，因此可将其压缩存储到一维数组中，并找到每个非零元素在一维数组中的对应关系。然而，在实际应用中，还经常会遇到另一类矩阵，在这类矩阵中，非零元素比零元素少，且分布没有规律，这类矩阵被称为稀疏矩阵，其压缩存储比特殊矩阵要更为复杂。详见 5.3 节所述。

**【例 5-3】** 推导上、下三角矩阵的求址公式。

上三角矩阵：

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ c & a_{2,2} & \cdots & a_{2,n} \\ \vdots & & \ddots & \vdots \\ c & \cdots & \cdots & a_{n,n} \end{pmatrix}$$

下三角矩阵：

$$\begin{pmatrix} a_{1,1} & c & \cdots & c \\ a_{2,1} & a_{2,2} & \cdots & c \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix}$$

在三角矩阵中，重复元素  $c$  可共享一个存储空间，其余元素和对称矩阵一样，正好有

$(n+1)/2$  个。因此, 三角矩阵可压缩存储到数组  $M[1..n*(n+1)/2+1]$  中, 其中,  $c$  若非零, 则存放到数组的最后一个下标变量中。在上三角矩阵中, 主对角线上的第  $t$  行 ( $1 \leq t \leq n$ ) 有  $n-t+1$  个元素, 按行优先顺序存放上三角阵中的元素  $a_{ij}$  时,  $a_{ij}$  之前的前  $i-1$  行共有  $(2n-i+2)/2$  个元素。在第  $i$  行上,  $a_{ij}$  是该行的第  $j-i+1$  个元素。  $M[k]$  和  $a_{ij}$  的对应关系是:

$$k = \begin{cases} \frac{(i-1)(2n-i+2)}{2} + j - i + 1 & \text{当 } i \leq j \text{ 时} \\ \frac{n(n+1)}{2} + 1 & \text{当 } i > j \text{ 时} \end{cases}$$

当  $i > j$  时,  $a_{i,j} = c$ ,  $c$  存放在  $M[\frac{n(n+1)}{2} + 1]$  中。

下三角矩阵的存储与对称矩阵类似。  $M[k]$  与对称矩阵类似。  $M[k]$  和  $a_{i,j}$  的对应关系是:

$$k = \begin{cases} \frac{i(i-1)}{2} + j & \text{当 } i \geq j \text{ 时} \\ \frac{n(n+1)}{2} + 1 & \text{当 } i < j \text{ 时} \end{cases}$$

### 5.3.2 稀疏矩阵

#### 1. 定义

对于稀疏矩阵, 没有确切的定义, 最常见的有以下两种:

定义 1 假设矩阵  $A_{mn}$  中有  $s$  个非零元素, 若  $s$  远远小于矩阵元素的个数 ( $s \ll m \times n$ ), 并且非零元素的分布没有规律, 则称  $A$  为稀疏矩阵。

定义 2 假设在  $m \times n$  的矩阵中, 有  $t$  个元素不为零。令  $\delta = t/(m \times n)$ , 称  $\delta$  为矩阵的稀疏因子。通常认为,  $\delta \leq 0.05$  时为稀疏矩阵。

在 ADT5-2 中, 列举了几种常见的矩阵运算。

#### ADT 5-2 稀疏矩阵的抽象数据类型描述

```
ADT SparseMatrix {
    数据集合:  $D = \{ a_{i,j} \mid i = 1, 2, \dots, m; j = 1, 2, \dots, n; a_{i,j} / \text{ElemSet}, m \text{ 和 } n \text{ 分别称为矩阵的行数和列数} \}$ 
    数据关系:  $R = \{ \text{Row}, \text{Col} \}$ 
        Row =  $\{ \langle a_{i,j}, a_{i,j+1} \rangle \mid 1 \leq i \leq m, 1 \leq j \leq n-1 \}$ 
        Col =  $\{ \langle a_{i,j}, a_{i+1,j} \rangle \mid 1 \leq i \leq m-1, 1 \leq j \leq n \}$ 
    数据操作:
        Create_SMatrix(&M); //初始化稀疏矩阵
            输出: 创建稀疏矩阵 M。
        Destroy_SMatrix(&M); //撤销稀疏矩阵
            输入: 稀疏矩阵 M。输出: 销毁稀疏矩阵 M。
        Print_SMatrix(M); //打印稀疏矩阵
            输入: 稀疏矩阵 M。输出: 打印稀疏矩阵 M。
        Copy_SMatrix(M, &T); //复制稀疏矩阵
            输入: 稀疏矩阵 M。输出: 由稀疏矩阵 M 复制得到 T。
}
```

```
Add_SMatrix(M, N, &Q); //行数和列数对应相等的稀疏矩阵相加
    输入：行数和列数对应相等的稀疏矩阵 M 与 N。
    输出：稀疏矩阵的和 Q=M+N。
Sub_SMatrix(M, N, &Q); //行数和列数对应相等的稀疏矩阵相减
    输入：行数和列数对应相等的稀疏矩阵 M 与 N。
    输出：稀疏矩阵的差 Q=M - N。
Mult_SMatrix(M, N, &Q); //行数和列数对应相等的稀疏矩阵相乘
    输入：行数和列数对应相等的稀疏矩阵 M 与 N。
    输出：稀疏矩阵的乘积 Q=M×N。
Transpose_SMatrix(M, &T); //稀疏矩阵转置
    输入：稀疏矩阵 M。输出：稀疏矩阵 M 的转置矩阵 T。
```

}ADT SparseMatrix

由于用数组存储稀疏矩阵中的元素时，仅有少部分空间存储了实际的数据元素，因此，为节省存储空间，通常采用一种压缩的存储方法来表示稀疏矩阵的内容。由于稀疏矩阵中非零元素分布没有一定规律，必须同时记下它所在行和列的位置( $i, j$ )，因此可用一个三元组( $i, j, a_{i,j}$ )来唯一确定矩阵  $A$  的一个非零元素，即稀疏矩阵可由表示非零元素的三元组及其行、列数唯一确定。例如，下列三元组表((1, 2, 9), (2, 1, 5), (3, 4, 7), (4, 1, 4))加上(4, 4)这一对行、列数便可作为图 5.6 所示矩阵  $A$  的另一种描述。

$$A_{4 \times 4} = \begin{pmatrix} 0 & 9 & 0 & 0 \\ 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 \\ 4 & 0 & 0 & 0 \end{pmatrix}$$

图 5.6 稀疏矩阵  $A$

## 2. 三元组顺序表

假设非零元素的三元组以按行优先的原则顺序排列，一个稀疏矩阵可转换成一个对应的线性顺序来表示，其中每个元素由一个上述的三元组构成，这个线性表就称为三元组表，它是一种具体的线性表顺序存储结构，因此可看成是线性表顺序存储结构 (LinearListSqu) 的派生类，只是元素类型  $T$  要具体定义，程序 5-7 和程序 5-8 分别给出了三元组的元素类型定义和类描述。

程序 5-7 三元组的元素类型定义

```
class TriTuple //定义三元组的元素类型
{
public:
    long row, col; float val; //行号、列号、值
    bool operator==(TriTuple &il) //重载恒等算符
        return(row==il.row && col==il.col && val==il.val);
        //两个三元组元素的行号、列号、值全部对应相等时才算相等
};
TriTuple& operator=(TriTuple &il) //重载赋值算符
{
    This->row=il.row; this->col=il.col; this->val=il.val; return il;
};
};
```

这里对恒等(==)和赋值(=)算符进行了重载，这是因为，在 TLinearListSqu (下面将作为三元组类型的父类)中要求元素类型  $T$  (以可变类型出现)支持恒等与赋值运算。为



了支持父类的输出操作（Output），还需要对标准输出算符进行重载：

```
ostream& operator <<(ostream& oo, TriTuple &il)
{ oo<<"("<<il.row<<"", "<<il.col<<"", "<<il.val<<""); return oo; };
```

程序 5-8 三元组表的类描述

```
class TriTupleArray :public LinearListSqu<TriTuple>
{ long rows, cols; //总行数、列数
  float theZero; //具体的“0”元素值，而不是数值 0
  long AddNew(long i, long j, float x);
  //在三元组表中加入一个新元素(i, j, x)
public:
  TriTupleArray(long rows, long cols);
  TriTupleArray(long rows, long cols, long mSize);
  bool Get(long i, long j, float &x); //读出元素(i, j)的值
  bool Set(long i, long j, float x); //将元素(i, j)的值置为 x
  long GetRowsCols(long &r, long &c); //返回最大行号和列号
  bool Delete(long i, long j); //删除元素(i, j)
  int Trans1(TriTupleArray &tu); //转置
  int Trans2(TriTupleArray &tu); //转置
  virtual void Output(ostream &out) const; //输出函数
};
```

3. 矩阵的转置

矩阵的转置，就是使 *i* 行 *j* 列的元素与 *j* 行 *i* 列的元素对换位置。若矩阵是用二维数组表示的，则转置操作很简单。设 *a* 是一个用二维数组表示的 *m*×*n* 矩阵，其转置的结果存于二维数组 *b* 中，它是一个 *n*×*m* 矩阵。具体的转置算法参见程序 5-9。

程序 5-9 转置算法

```
int a[m][n], b[n][m];
...
for (i=0; i<m; i++)
    for (j=0; j<n; j++) b[j][i]=a[i][j];
```

如果矩阵是用三元组表表示的，可利用 Get 和 Set 操作，则转置操作与上面的类似，基本形式为：

```
for(i=0; i<m; i++)
    for (j=0; j<n; j++)
        b.Set(j, i, a.Get(i, j));
```

这个例子表明，三元组表设置了 Get 和 Set 操作后，其所用的方式就与二维数组相同了，完全屏蔽了三元组表的存储结构。为了提高程序效率，可直接实现转置（不使用 Get 和

Set), 则转置的实现没有这样直接。分析转置操作, 其实质是将每个元素的行号与列号互换。由于在三元组表中, 元素是按行序(或列序)排列的, 所以行号和列号互换后, 还要调整元素位置, 使其仍保持按行序(或列序)排列。实现这一点的一个显然的做法是: ①将每个元素的行号和列号分别互换; ②对三元组表排序, 使其中元素按行序(或列序)排列。

此算法的时间复杂度为  $O(n)+O(n \log(n))$ , 这里  $n$  为三元组表中元素个数。 $O(n \log(n))$  是基于比较运算的排序算法的平均最好时间复杂度的, 在简单排序情况(如冒泡排序等)下, 时间复杂度可上升为  $O(n^2)$ 。如果要降低时间复杂度, 一个显然的做法是, 免去排序操作, 在进行元素的行号和列号互换的过程中, 同时实现行序(列序)排列。也就是(假定将  $a$  转置结果存于  $b$  中): ① 将  $a$  的每个元素从三元组表中取出, 交换它们的行号值与列号值; ② 将所取出的三元组元素存入目标三元组表  $b$  中适当位置, 使三元组表  $b$  保持行序/列序。

下面假定三元组中元素按行序排序, 即原矩阵  $A$  与转置结果均按行序排列。对于列序排列情况, 处理方法类似。注意, 由于这里假定三元组表内元素按行序排列, 且转置后元素的行列号互换了, 所以在转置后的三元组表中, 元素相当于按原三元组表中元素的列序排列。这里给出一个稀疏矩阵  $A$  与其三元组形式  $a$ , 以及它们的转置形式  $B$  与其三元组形式  $b$  (见图 5.7)。

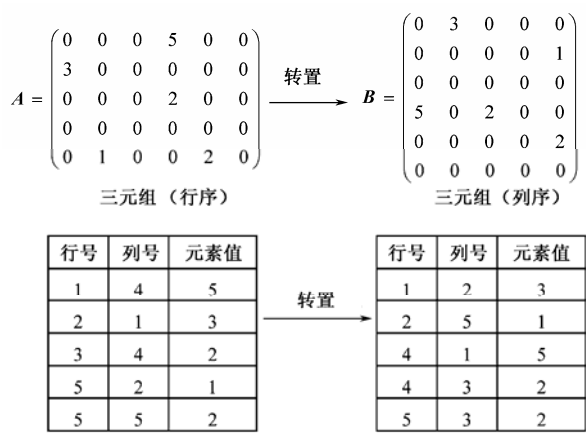


图 5.7 矩阵转置

对三元组转置, 算法如下。

该算法可概括为“直接取, 顺序存”, 即第 1 次从  $a$  中选取一个适当元素放置到  $b$  中的第 1 个位置, 第 2 次从  $a$  中选取一个适当元素放到  $b$  中的第 2 个位置, …… , 如此进行, 依次生成  $b$  中的第 1、2…个元素。由于转置后列号变行号, 故转置后元素按原列号顺序排列(以保持行序)。因此, 该算法的实现, “直接取”是指在  $a$  中按列号的大小取元素, 即依次取第 1、2…列元素。当某列上有多个元素时(列号为  $i$  的元素有多个时), 按它们的行号顺序取。“顺序取”是指将所取出的元素依次放到  $b$  中的最近空位处(即最后一个元素的下个位置)。其过程可描述如下:

```
pb=1; //pb 为 b 中当前空位置中编号最小的位置的指示器
for (col=最小列号; col<=最大列号; col++)
{   在 a 中按从头到尾的顺序, 查找有无列号为 col 的三元组;
    若有, 则将其行列交换后依次存入 b 中 pb 所指位置;
```

```

        pb 加 1;
    }

```

注意，由于已假定三元组表元素按行序排列，所以当在上面的程序中列号等于 col 的元素有多个时，它们中必然是行号较小者较先出现，因此，可保证列号（在转置后的三元组表中是行号）相同时按行号（在转置后的三元组表中是列号）排列。据此可写出完整的算法程序，见程序 5-10。

程序 5-10 三元组表的转置算法 I

```

int TriTupleArray::Trans1(TriTupleArray &tu)
{
    //将本类所对应的对象转置，结果存入 tu
    long pa, pb, col; long mCols, mRows;
    if (length<1) return 0;
    if (tu.ResizeRoom(length)<=0)
        return 0; //调整 tu 的空间，使其等于本对象中的元素个数
    GetRowsCols(mRows, mCols); //求最大行号和列号
    pb=0;
    for (col=0; col<=mCols; col++)
    {
        for (pa=0; pa<length; pa++)
            if (element[pa].col==col)
            {
                tu.element[pb].row=element[pa].col; tu.element[pb].col=element[pa].row;
                tu.element[pb].val=element[pa].val; pb++;
            }
    }
    tu.cols=mRows; tu.rows=mCols; tu.length=length;
    return length;
}

```

## 5.4 十字链表

### 5.4.1 存储方式

三元组表采用顺序方法来存储稀疏矩阵中的非零元素，它不适合于非零元素个数和位置在操作过程中变化较大的矩阵。例如，在两个矩阵进行相加操作时，会改变非零元素的个数，这样将导致大量结点的移动。此时，采用链式存储结构来表示三元组的线性表更为合适。

一般采用十字链表的链接存储方式，即稀疏矩阵中的每个非零元素用一个包含 5 个域的结点表示，结点结构如图 5.8 所示，其中除了表示非零元素所在的行、列和值的三个域以外，还有两个分别指向同一行中下一个非零元素的 right 域和同一列中下一个非零元素的 down 域。因此，同一行的非零元素通过 right 域链接成一个线性链表，同一列的非零元素通过 down 域链接成一个线性链表。每个非零元素既是某个行

row	col	val
down	right	

图 5.8 十字链表的结点结构

链表中的一个结点，又是某个列链表中的一个结点，整个矩阵构成了一个十字交叉的链表，故称这样的存储结构为十字链表。这里以稀疏矩阵为背景介绍十字链表，不过十字链表的应用远不止稀疏矩阵，一切具有正交关系的结构，都可用十字链表存储。

【例 5-4】 设有一个如下形式的矩阵，它所对应的十字链表如图 5.9 所示。

$$\begin{pmatrix} 0 & 0 & 5 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 4 & 0 \\ 0 & 1 & 0 & 2 & 0 \end{pmatrix}$$

从图 5.9 可看到，矩阵第 1 行有一个非 0 元素 5，在十字链表中表示为结点(0, 2, 5)；总头结点中的 4 和 5 表示矩阵共有 4 行 5 列；行头结点用虚线画出，表示每行头结点与每列头结点为同一结点，如第 1 行/列头结点中的 1 和 2 表示：第 1 行有 1 个非 0 元素，第 1 列有 2 个非 0 元素。

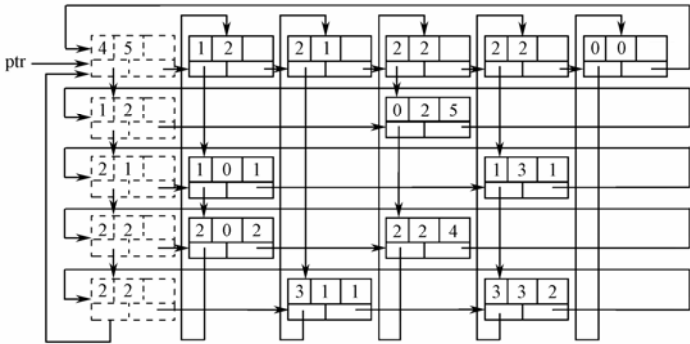


图 5.9 十字链表示意图

### 5.4.2 十字链表对象

程序 5-11 和程序 5-12 分别给出了十字链表结点类和链表类的 C++语言描述，同时，为兼容比较运算、赋值运算和标准输出运算等，还定义了相应运算符重载。

程序 5-11 十字链表结点类

```
template <class T>class CrossNode //十字链表结点类型
{
public:
    long row, col; //行号、列号
    union { //让 val 和 next 共享一块存储空间，即一块空间，两个名字
        T val; //元素值
        CrossNode *next; //用于将头结点链为链表
    };
    CrossNode *down, *right; //列 / 行指针
    operator==(CrossNode &oo) //重定义恒等运算
    { return (row==oo.row && col==oo.col && val==oo.val); };
    CrossNode& operator=(CrossNode &oo) //重定义赋值运算
    { this->row=oo.row; this->col=oo.col; this->val=oo.val; return oo; };
```

```
};
```

## 程序 5-12 十字链表类

```
template <class T>
class CrossLink //十字链表对象
{
    CrossNode<T> *head; //指向十字链表总头结点的指针
    //根据该指针,可访问到十字链表中的各元素
    long rows, cols; //矩阵的最大行号和列号
    T theZero; // "0"元素的值,存储 0 值的变量,主要为了按引用方式返回 0 元素值使用
    void ReleaseAll(void); //释放所有结点
    CrossNode<T> *Insert(CrossNode<T> *pNode);
    //将 pNode 作为 i 行 j 列元素结点插入到十字链表, i 与 j 的值在 pNode 中
    //该函数主要用于 Set 函数
    //当 Set 函数为一个不存在的结点(即 0 元素)置值时,调用该函数插入一个结点
    CrossNode<T> *Insert(T& x, long i, long j);
    CrossNode<T> *Delete(long i, long j); //将 i 行 j 列元素结点删除
    //该函数主要用于 Set 函数
    //当 Set 函数为一个非 0 元素置 0 值时,调用该函数将对应的结点删除
public:
    long len; //十字链表中元素结点的个数(不含各种头结点)
    //对稀疏矩阵,是非 0 元素的个数

    CrossLink();
    CrossLink(long rows, long cols);
    ~CrossLink();
    void Init(long rows, long cols);
    //初始化操作。用于创建一个具有 rows 行和 cols 列的空的十字链表
    //构造函数基本上是直接调用 Init()实现的
    T& Get(long i, long j); //返回 i 行 j 列元素的值的引用
    //若该元素结点不存在(0 元素),则返回 0 元素值
    CrossNode<T> *GetNode(long i, long j); //返回 i 行 j 列元素结点的指针
    //若该元素不存在,则返回空
    CrossNode<T> *Set(long i, long j, T &x); //将 i 行 j 列元素的值置为 x
    //若 i 行 j 列元素原为 0 元素,则该函数的执行将在十字链表中插入一个新结点
    //若 x 值为 0,且该结点存在,则该函数的执行将删除该结点
    CrossNode<T> *GetHead(long k); //返回行 / 列号为 k 的行 / 列头链表上的结点的指针
    void print(); //访问输出链表各元素
};
```

### 5.4.3 基本操作的实现

#### 1. 初始化操作

初始化操作见程序 5-13。该函数生成一个

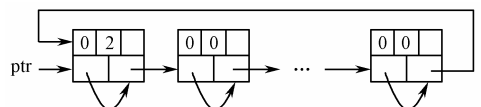


图 5.10 十字链表

具有 rows1 行和 cols1 列的空十字链表。空的十字链表只含有各行/列的头结点和总头结点。由于序号相同的行与列共享头结点，故总共生成  $1 + \text{Max}(\text{rows1}, \text{cols1})$  个结点，每个结点以 next 链接，形成以总头结点为头的循环链表，其形式如图 5.10 所示。

程序 5-13 初始化操作

```
template <class T>
void CrossLink<T>::Init(long rows1, long cols1)
{   if (rows1<=0 || cols1<=0)   return ;
    CrossNode<T> *p, *h;
    long rc; rc=rows1;
    if (rc<cols1) rc=cols1; //令 rc 为 rows1 与 cols1 中最大者
    if (head!=0) ReleaseAll( ); //若原表中有结点，则先释放之
    h=new CrossNode<T>; //申请一个新结点，作为总头结点
    if (h==0)   throw NoMem( );
    h->next=h; h->row=rows1; h->col=cols1;
    for (long i=0; i<rc; i++)
    { //每次生成一个新结点并插入在总头结点后面
        p=new CrossNode<T>;
        if (p==0)   throw NoMem( );
        p->row=p->col=-1;
        //新结点是行/列的首结点，row 及 col 分别代表该行/列有多少个非 0 元素
        //初值赋为-1
        p->down=p; p->right=p; p->next=h->next; //将 p 插入在总头结点后面
        h->next=p;
    }
    rows=rows1; cols=cols1; head=h;
}
```

## 2. 插入操作

如果将十字链表仅仅作为稀疏矩阵，则插入操作属于内部操作，它将行号和列号已知的结点插入到十字链表中指定位置。因为每个结点同时属于两个链表（行链表与列链表），所以插入一个结点时，应分别插入到这两个链表中。有两个插入版本：一个是将存在的链结点插入；另一个是已知元素值进行插入，它通过调用前者实现。插入操作见程序 5-14。

程序 5-14 插入操作

```
template<class T>
CrossNode<T> *CrossLink<T>::Insert(CrossNode<T> *pNode)
{ //将 pNode 插入到十字链表中，pNode 中包含行号和列号
    long i, j;
    CrossNode<T> *p, *p0;
    i=pNode->row; j=pNode->col;
    p0=GetHead(i+1); //得到行 i 的头结点，然后在行 i 中插入 pNode
    if (p0==NULL)   throw NoMem( );
```

```

p=p0;
while (p->right!=p0 && p->right->col<j)p=p->right; //在行 i 中找插入位置 p
pNode->right=p->right; pNode->right=pNode; //在 p 后插入 pNode
if (p0->row==-1)p0->row=p0->row+2; //修改该行的非 0 元素个数
else p0->row++;
p0=GetHead(j+1); //得到列 j 的头结点, 然后在列 j 中插入 pNode
if (p0==NULL) throw NoMem( );
p=p0;
while (p->down!=p0 && p->down->row<i) //在列 j 中寻找插入位置 p
    p=p->down;
pNode->down=p->down; pNode->down=pNode; len++; //在 p 后插入 pNode
if (p0->col==-1)p0->col=p0->col+2; //修改该列的非 0 元素个数
else p0->col++;
return head;
}

template <class T>
CrossNode<T> *Insert(T& x, long i, long j)
{//插入一个以 x 为元素值的结点, 行号和列号分别为 i 和 j
    TCrossNode<T> *p;
    p=new(nothrow) TCrossNode<T>; //申请一个结点
    if (p==NULL) throw NoMem( );
    p->row=i; //给结点置值 p->col=j;
    p->val=x; Insert(p); //调用另一版本的 Insert 将 p 插入
    return p;
}

```

### 3. 删除操作

与插入操作类似, 如果将十字链表仅仅作作为稀疏矩阵, 则删除操作也属于内部操作, 它将行号和列号已知的结点从十字链表中删除。因为每个结点同时属于两个链表(行链表与列链表), 所以删除一个结点时, 应分别从这两个链表中删除。Get 类和 Set 类请读者自行实现。

#### 5.4.4 十字链表相加法\*

设有两个按十字链表存储的矩阵  $A_{m \times n}$  和  $B_{m \times n}$ , 现考虑将  $B$  加到  $A$  上, 即相加操作  $A \leftarrow A+B$ , 也就是对应元素相加  $a_{i,j}+b_{i,j}$ 。下面考虑采用逐行相加的方法, 即逐次将  $B$  的各行加到  $A$  上, 实现过程可描述为(伪码):

```

for (i=1; i<=m; i++) { pHB=B.GetHead(i); }
//求 B 中第 i 行头结点, 将 B 的第 i 行加到 A 的第 i 行上, 问题转化为“如何将 B 的第 i 行加
//到 A 的第 i 行上”

```

该项操作的实现与多项式加法十分类似, 见程序 5-15。

```

pHA=A.GetHead(i); //求得 A 的第 i 行头结点
pA=pHA->right; //pA 指向 A 中第 i 行中当前待处理的结点
pB=pHB->right; //pB 指向 B 中第 i 行中当前待处理的结点
pA0=pHA; pB0=pHB; //pA0 与 pB0 分别为 pA 与 pB 的前驱
while (pB->right!=pHB) //处理 B 的第 i 行中各结点
{ if (pA->col<pB->col) { pA0=pA; pA=pA->right; } //pA 后移一步
  else if (pA->col>pB->col)
  {      从 pB 复制一个结点 p;
        将 p 插入到 A 的第 i 行链表中 pA 之前, 即 pA0 之后;
        将 p 插入到 A 的 pB->col 列链表中适当位置;
        pB0=pB; pB=pB->right;
    }
  else //pB->col==pA->col
  {      pA->val=pA->val + pB->val;
        if (pA->val !=0 ) //pA 与 pB 分别后移一步
        { pA0=pA; pA=pA->right; pB0=pB; pB=pB->right; }
        else
        {      将 pA 从十字链表中摘除; 释放 pA;
              pA=pA0->right; //令 pA 指向被删结点的下一结点
              pB=pB->right; //pB 后移一步
        } else
        } //else
    } //while

```

在上列算法描述中, 尚有一些操作需细化, 下面分别进行讨论。

从 pB 复制一结点 p:

```
p=new TCrossNode; p->row=pB->row; p->col=pB->col; p->val=pB->val;
```

将 p 插入到 A 的第 i 行链表中 pA 之前 (pA0 之后)

```
p->right=pA0->right; pA0->right=p;
```

将 p 插入到 A 的 pB->col 列链表适当位置:

```

q0=q.A.GetHead(pB->col); //求得 A 的 pB->col 列的头结点
while (q->down!=q0 && q->down->row<p->row)
    q=q->down; //在 pB->col 列链表中查找插入位置 q
p->down=q->down; q->down=p; //将 p 插入到 q 之后
pA0->right=pA->right; //将 pA 从行链表中删除
qo=q.A.GotHead(pA->col); //求得 pA 所在列的头结点
while (q->down!=qo && q->down->row<pA->row)
    q=q->down; //查找 pA 的前驱 q->down=pA->down; //将 pA 从列链表中摘除
free(pA);

```



**【例 5-5】** 读入一个矩阵并建立它的十字链表示，使用一个辅助的全程量数组 `hdnode`。  
算法如下：

---

```
struct matrixnode
{
    matrixnode *down; matrixnode *right;
    bool head; matrixnode *true;
    int value, row, cot;
}

void mread(matrixnode *a)
{
    int i, m, n, p, r, row, cot, val, currentrow;
    matrixnode *x, *last;
    cin>>n>>m>>r; //矩阵的维数
    if (m>n) p=m; else p=n; //p 为表头结点的个数
    for (i=0; i<p; i++) //表头结点初始化
    {
        x=new matrixnode; hdnode[i]=x; x->right=x; x->head=true; x->next=x;
    }
    currentrow=0; last=hdnode[0]; //当前行中的最后结点
    for (i=0; i<r; i++) //输入三元组
    {
        cin>>rrow>>ccol>>val;
        if (rrow>currentrow)
        {
            last->right=hdnode[currentrow]; culrentrow=rrow; //关闭当前行
            last=hdnode[rrow];
        }
        x=new matrixnode; //关于新的三元组的结点
        x->head=false; x->row=rrow; x->col=ccol; x->value=vad;
        last->right=x; last=x; //链到行表中去
        hdnode [ccol] ->next=x; //链到列表中去
    } //关闭最后行
    if (r>0) last->right=hdnode[currentrow];
    for (i=0; i<m; i++) //关闭所有的列表
        hdnode[i] ->next->down=kdnode[i]; //建立表头结点链表的表头结点
    a=new matrixnode; //把表头结点链接在一起
    a->head=false; a->row=n-1; a->col=m-1;
    for (i=0; i<p-1; i++) hdnode[i] ->next=hdnode[i+1];
    if (p=0) a->right=a; //十字链表是空表
    else { hdnode[p-1] ->next=a; a->right=hdnode[0]; }
} //算法时间复杂度  $q(\max \{in, m\} + r) = O(n+m+r)$ 
```

---

## 5.5 广 义 表

### 5.5.1 广义表的定义

广义表 (List, 又称列表) 是线性表的推广, 它放松了对表元素的原子限制, 允许表元

素具有其自身结构。广义表是  $n$  ( $n \geq 0$ ) 个元素  $a_1, a_2, \dots, a_i, \dots, a_n$  的有限序列,  $a_i$  或者是一个原子, 或者是一个广义表。广义表通常记作  $LS = (a_1, a_2, \dots, a_i, \dots, a_n)$ , 其中,  $LS$  是广义表的名字,  $n$  为它的长度, 如果  $a_i$  也是广义表, 则称它为  $LS$  的子表。广义表通常用圆括号括起来, 用逗号分隔其中的元素; 为了区分原子和广义表, 书写时用大写字母表示广义表, 用小写字母表示原子; 若广义表  $LS$  非空 ( $n \geq 1$ ), 则  $a_1$  是  $LS$  的头, 其余元素组成的表  $(a_2, \dots, a_n)$  称为  $LS$  的表尾; 广义表是递归定义的。

- 【例 5-6】** ①  $L1 = ()$  表示:  $L1$  是一个空表, 它的长度为 0, 深度为 1。  
②  $L2 = (\epsilon)$  表示: 列表  $B$  只有一个原子  $\epsilon$ ,  $L2$  的长度为 1, 深度也为 1。  
③  $L3 = (\alpha, (\beta, \gamma, \delta))$  表示: 列表  $L3$  的长度为 2, 深度为 2, 两个元素分别为原子  $\alpha$  和子表  $(\beta, \gamma, \delta)$ 。  
④  $LS = (L1, L2, L3)$  表示: 列表  $LS$  的长度为 3, 深度为 1, 3 个元素都是列表。将子表的值带入后, 则有  $LS = (( ), (\epsilon), (\alpha, (\beta, \gamma, \delta)))$ 。  
⑤  $L4 = (\alpha, L4)$  表示: 这是一个递归的表, 它的长度为 2, 深度为  $\infty$ 。 $L4$  相当于一个无限的列表  $L4 = (\alpha, (\alpha, (\alpha, \dots)))$ 。

### 5.5.2 广义表的抽象数据类型定义

广义表的抽象数据类型定义见 ADT 5-3。

ADT 5-3 广义表的抽象数据类型描述

```
ADT GList {
    数据集合:  $D = \{ e_i \mid i=1, 2, \dots, n; n \geq 0; e_i / \text{AtomSet 或 } e_i / \text{GList}; \}$ 
    数据关系:  $R = \{ \langle e_{i-1}, e_i \rangle \mid e_{i-1}, e_i / D, 2 \leq i \leq n \}$ 
    数据操作:
        Init_GList(&L); //初始化广义表
            输出: 空的广义表 L。
        Create_GList(&L, S); //由串生成广义表
            输入: 广义表的书写形式串 S。输出: 由 S 创建的广义表 L。
        Destroy_GList(&L); //撤销广义表
            输入: 广义表 L。输出: 删除广义表 L。
        Copy_GList(&T, L); //复制广义表
            输入: 广义表 L。输出: 由广义表 L 复制得到广义表 T。
        Length_GList (L); //求广义表的长度
            输入: 广义表 L。输出: 广义表 L 的长度, 即元素个数。
        Depth_GList (L); //求广义表的深度
            输入: 广义表 L。输出: 广义表 L 的深度。
        Empty_GList (L); //判断广义表是否为空
            输入: 广义表 L。输出: 判定广义表 L 是否为空。
        Get_Head(L); //获取广义表的头
            输入: 广义表 L。输出: 广义表 L 的头。
        Get_Tail(L); //获取广义表的尾
            输入: 广义表 L。输出: 广义表 L 的尾。
        InsertFirst_GL(&L, e); //在广义表第一个位置插入新元素
```

输入：广义表 L。输出：插入元素 e 作为广义表 L 的第一个元素。  
DeleteFirst\_GL(&L, &e); //删除广义表的第一个元素  
输入：广义表 L。输出：删除广义表 L 的第一个元素，用 e 返回其值。  
Traverse\_GL(L, visit( )); //遍历广义表  
输入：广义表 L。  
输出：遍历广义表 L，用函数 visit 处理每个元素。

}ADT Glist

广义表有如下重要结论。

- ① 列表的元素可以是子表，而子表的元素还可以是子表，故列表是一个多层次的结构。如图 5.11 所示的列表 LS，图中圆圈表示列表，方块表示原子。
- ② 列表可为其他列表所共享。
- ③ 列表可以是一个递归的表，即列表也可以是其本身的一个子表。

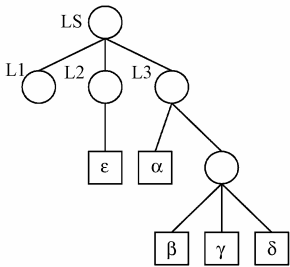
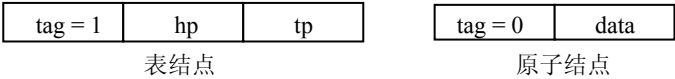


图 5.11 列表的图形表示

### 5.5.3 广义表的存储结构

#### 1. 头尾链表存储表示

##### (1) 结点结构



其中，tag 是标志域，tag=1 表示表结点，tag=0 表示原子结点；hp 指示表头的指针域；tp 指示表尾的指针域；data 是值域。

##### (2) 存储结构描述（见程序 5-16）

程序 5-16 广义表的存储结构

```
template<class T>
class GLNode
{
public:
    int tag; //公共部分，用于区分原子结点和表结点
    tag==0; //原子
    tag==1; //子表
    union{
        T atom; //存放数据
        GLNode<T> *hp, *tp; //表结点的指针域，hp 和 tp 分别指向表头和表尾
    };
};
```

(3) 特点

- ① 除空表的表头指针为空外，对任何非空表，其表头指针均指向一个表结点；
- ② 容易分清列表中原子和子表所在层次；
- ③ 最高层的表结点个数即为列表的长度。

(4) 图形表示

举例如下。

【例 5-7】  $LS=(L1, L2, L3)$ ，其中  $L1=()$ ， $L2=(\epsilon)$ ， $L3=(\alpha, (\beta, \gamma, \delta))$ ，如图 5.12 所示。

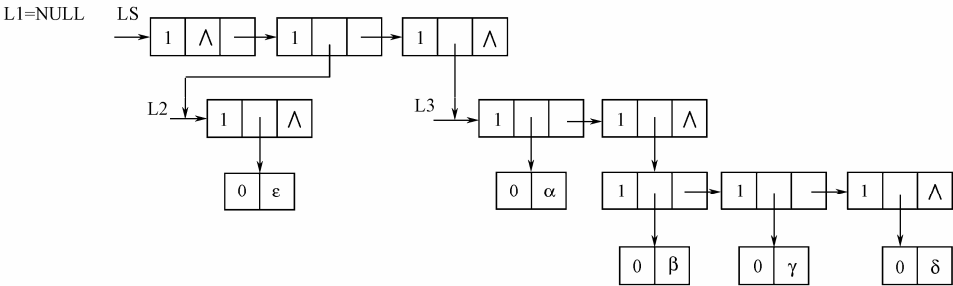


图 5.12 广义表 D 的存储结构示例

【例 5-8】  $L4=(\alpha, L4)$ ，如图 5.13 所示。

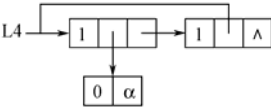
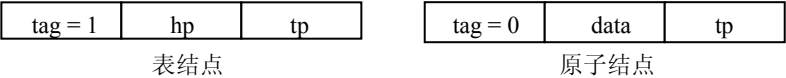


图 5.13 广义表 L4 的存储结构示例

2. 扩展性链表存储表示

(1) 结点结构



其中，tag 是标志域，tag=1 表示表结点，tag=0 表示原子结点；hp 指示表头的指针域；tp 指示表尾的指针域；data 是值域。

(2) 扩展性链表存储结构（见程序 5-17）

程序 5-17 扩展性链表存储结构

```
template<class T>
class GLNode
{
public:
    int tag; //公共部分，用于区分原子结点和表结点
    tag==0; //原子
    tag==1; //子表
    union{
        T atom; //存放数据
```

```

        GLNode<T> *hp; //表结点的表头指针
    };
    GLNode<T> *tp; /// 相当于线性链表的 next, 指向下一个元素结点
};

```

### (3) 广义表类定义

广义表是一种扩展的线性链表，程序 5-18 给出相关的类定义，读者可自行扩充。

程序 5-18 广义表类定义

```

template<class T>
class GenList
{
    GLNode<T> *first; //建立广义表的表头指针
    int Depth(GLNode<T> *gl); //计算非递归表(*gl)的深度
    GLNode *Copy(GLNode<T> *gl);
    int Equal(GLNode<T> * s, GLNode<T> *t); //比较两个表(*s, *t)是否相等
    void Delete(GLNode<T> *gl); //释放广义表
public:
    GenList( );
    virtual ~GenList( );
    int Depth( ); //计算一个非递归表的深度
    GLNode<T> & Head( ); //返回广义表中的第一个结点的 value
    GLNode<T> *Tail( ); //返回表中第二个结点的指针
    GLNode<T> *First( ); //返回表中第一个结点的指针
    GLNode<T> Next(GLNode<T> &elem); //返回 elem 所指结点后继结点的指针
};

```

### (4) 图形表示

前面例 5-7 和例 5-8 中的广义表的存储结构可表示为如图 5.14 和图 5.15 所示的结构。

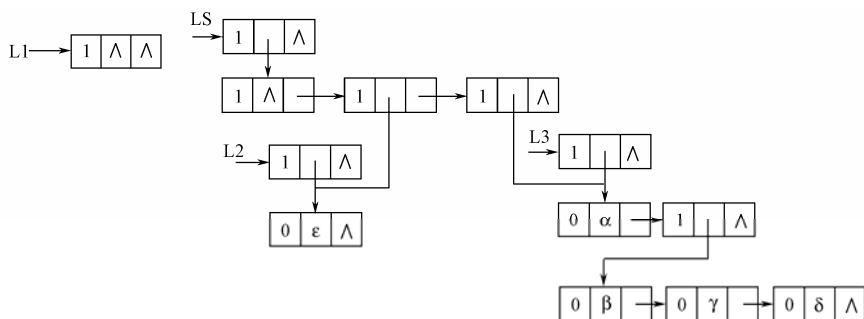


图 5.14 例 5-7 的另一种链表表示

**【例 5-9】** 用链表来表示广义表是很方便的。链表的一个结点表示广义表中的一个元素，具体格式为标志域：若该点表示一个原子，则为 false；若表示一个广义表，则为 true。dlink/data 为指向子表的指针或原子的名称，link 域指向下一个结点的指针。若无下一个结点，则

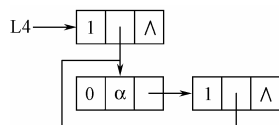


图 5.15 例 5-8 的另一种链表表示

为空。定义如下：

tag	dlink	data	link	tag
-----	-------	------	------	-----

```
struct listnode
{
    listnode *link;
    bool tag;
    char data;
    listnode *dlink;
};
```

现只给出 `equal(a, b)` 的实现，并为了简化程序和容易理解，仅限于考虑 `a` 和 `b` 是非递归的情形。程序如下：

```
bool equal(listnode *s, listnode *t)
//若 s 和 t 是“等同”的，则返回真；否则返回假，其中 s 和 t 是非递归的广义表
{
    bool x; x=false;
    if ((s==NULL)&&(t==NULL)) x=true;
    else if ((s != NULL)&&(t != NULL))
    {
        if (s->tag==t->tag)
        {
            if (!s->tag) { if (s->data==t->data) x=true; else x=false; }
            else x=equal(s->dlink, t->dlink); }
    }
    if (x) x=equal(s->link, t->link);
}
```

# 本章总结

## 1. 学习要点

本章主要介绍二维（多维）数组的逻辑结构形式化定义和顺序存储方式，特殊矩阵和稀疏矩阵的压缩存储方式，广义表的逻辑结构、存储结构以及运算和递归算法。主要学习要点如下：① 二维（多维）数组的定义和顺序存储结构的实现方法及元素存储位置计算公式，特殊矩阵及稀疏矩阵压缩存储的基本思想和压缩存储的实现方法；② 在压缩存储下标变换方法以及进行矩阵转置、相加、相乘运算所采用的处理方法（算法）；③ 广义表的表头、表尾分析方法及存储结构的实现。

## 2. 基本要求

- （1）深刻理解二维（多维）数组的逻辑结构定义和顺序存储及压缩存储的基本思想；
- （2）清楚二维（多维）数组的逻辑结构定义、读和写两种基本运算；
- （3）知道读、写两种运算的实现手段是下标变量和赋值；
- （4）掌握二维（多维）数组采用顺序存储结构表示时，元素位置的计算公式；
- （5）掌握特殊矩阵及稀疏矩阵压缩存储的实现方法和转置、乘法运算的基本算法；
- （6）掌握特殊矩阵压缩存储的基本思想 and 对称矩阵及三角矩阵压缩存储的下标与元素

存储位置的对应关系;

(7) 清楚稀疏矩阵的定义、三元组表和十字链表表示方法, 以及在这两种表示方法下的转置、相加、相乘运算的基本思想和算法;

(8) 领悟广义表的定义、存储结构、递归程序设计的基本思想。

### 3. 重点与难点

重点是: 数组的逻辑定义、压缩存储方法、广义表的存储结构及串的连接。求子串、定位、置换基本运算(操作)的实现算法。难点是: 特殊矩阵在压缩存储下元素位置的求址公式、稀疏矩阵的转置、乘法算法、广义表的链式存储结构。

## 习题 5

5-1 已知  $A$  是二维对称数组, 为节省存储单元, 只将上三角的元素存于内存中, 试推导元素位置的公式, 并比较下三角的情况。

5-2 若在  $m \times n$  的矩阵中有一个元素  $a[i, j]$  满足下述条件:  $a[i, j]$  既是第  $i$  行元素中的最小值, 又是第  $j$  列元素中的最大值(称为鞍点)。试写一个求矩阵鞍点的算法, 并分析所写的算法所需时间。

5-3 现有三个数组  $A[n+1]$ ,  $B[n+1][m]$  和  $C[n+1][n+2]$ , 试问各个数组能存放多少个元素。

5-4 有三维数组  $A(2, 3, 4)$ , 数组中元素长度为 4 个字节。试求元素  $A_{2,3,2}$  的相对地址。

5-5 假定一维数组的各个单元存储字符串中的一个字符, 分别就顺序存储和链表存储两种情况写出本章介绍的各种串运算的算法。

5-6 设  $A$  为一个具有  $n$  行  $n$  列的上三角方阵。若将此三角阵的非零元素按列存储在数组  $b[1..n(n+2)/2]$  中, 且  $a_{1,1}$  存放  $b[1]$  中, 写出此三角阵非零元素  $a_{i,j}$  ( $i \leq j$ ) 的寻址公式。

5-7 设  $A$  和  $B$  是两个  $n$  阶的下三角阵, 两个三角阵非零元素的总数为  $n(n+1)$ 。设计一个方案, 用数组  $C[1..n, 1..(n+1)]$  表示这两个三角阵。试写出从数组  $C$  确定  $A$  的元素  $a_{i,j}$  及  $B$  的元素  $b_{i,j}$  ( $1 \leq i, j \leq n$ ) 的程序过程。(提示: 将  $A$  的三角阵放在  $C$  的下三角形位置上, 将  $B$  的转置阵放在  $C$  的上三角形位置上。)

5-8 将  $n$  阶三角矩阵  $A$  按行优先的方法存放在  $b[1..3n-2]$  中,  $A$  的元素  $a_{1,1}$  存放在  $b[1]$  中, 试设计一个算法: 从数组  $b$  决定  $a_{i,j}$  ( $1 \leq i, j \leq n$ ) 的值。

5-9 广义的带状矩阵是  $A_{nab}$  是指  $n \times n$  阶阵  $A$  中的非零元素都落在主对角线下面的  $a-1$  条对角线上和主对角线上面的  $b-1$  条对角线及主对角线上, 如图 5.16 所示。试问: ①  $A_{nab}$  的带状区域上共有多少个元素。②在  $A_{nab}$  的带上,  $a_{i,j}$  元素中的下标  $i$  和  $j$  之间有什么关系。③用一维数组  $C$  连续地存放  $A_{nab}$  的带上的元素, 对于这种存储方式, 编写一个 C++ 程序 `vaue(n, a, b, i, j, c)`, 以取得矩阵  $A_{nab}$  的带上的元素  $a_{i,j}$  的值。

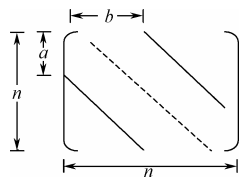


图 5.16 习题 5-9 的图

5-10 假设数组  $A[s_1..t_1, s_2..t_2, \dots, s_n..t_n]$  的每个元素占一个内存单元, 且  $A(s_1, s_2, \dots, s_n)$  的地址为  $\alpha$ , 试给出: ①数组按顺序存储时,  $A[i_1, i_2, \dots, i_n]$  的寻址公式; ②数组按列顺序存储时,  $A[i_1, i_2, \dots, i_n]$  的寻址公式。

5-11 如果用三元组表示稀疏矩阵的非零元素的方法来表示稀疏矩阵  $A$  和  $B$ , 试写一个求解  $A+B$  的程序过程, 并分析算法的执行时间。

5-12 编写将十字链表表示的矩阵  $A$  转置的程序算法, 转置结果为另一个十字链表, 并将该转置矩阵以三元组  $(i, j, \text{value})$  的形式输出。

5-13 已知稀疏矩阵  $W$ , 画出它的三元组表和十字链表。

5-14 写一个算法, 依次输入三元组表中的元素, 建立该稀疏矩阵的十字链表。

5-15 试对广义表 LS 进行 HEAD 和 TAIL 运算, 并画出其存储结构图。LS=(a, ((b, c), ( ), d), (((e))))。

5-16 如图 5.17 所示为广义表的存储结构图, 写出此图表示的广义表。

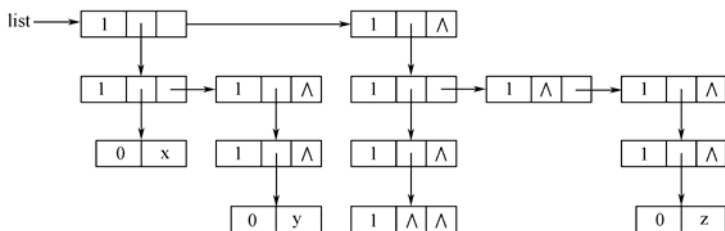


图 5.17 习题 5-16 的图

5-17 假设 L 为非递归并不带共享子表的广义表，试设计一个复制广义表 L 的算法。

5-18 有广义表  $A = ()$ ,  $B = (A, e, d)$ ,  $C = (a)$ ,  $D = (e, (a, b, B), A, C)$ 。要求: ①画出上述各广义表的图形; ②求表  $D$  的长度和深度; ③画出  $D$  表的链表表示。

5-19 考虑 4 个实数矩阵的乘积  $\mathbf{M}_1 \times \mathbf{M}_2 \times \mathbf{M}_3 \times \mathbf{M}_4$ 。其中, 矩阵  $\mathbf{M}_1$  的阶为  $10 \times 20$ ,  $\mathbf{M}_2$  为  $20 \times 50$ ,  $\mathbf{M}_3$  为  $50 \times 1$ ,  $\mathbf{M}_4$  为  $1 \times 100$ 。假设计算一个  $p \times q$  矩阵与一个  $q \times r$  矩阵的乘积, 需要  $p \times q \times r$  次标量操作 (通常的矩阵相乘算法就是如此)。试求出计算上述 4 个矩阵乘积的最优次序, 即按照这种次序进行计算可使标量操作的次数总和最小。如果要计算任意  $n$  个矩阵的乘积, 应如何求最优次序?

5-20 设计一算法, 将按行优先原则顺序存储的二维数组  $A_{nm}$  转置 ( $a_{i,j}$  与  $a_{j,i}$  交换), 要求转置结果仍占用原来的存储空间。

5-21 改写两个稀疏矩阵相乘的算法，计算  $C_{ln} = A_{lm}B_{mn}$ 。其中  $A$ 、 $B$  和  $C$  都采用带行（列）指针向量的单链表示方法或采用三元组表示法。

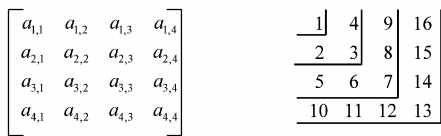
5-22 设有广义表  $J_1(J_2(J_1 \text{ a } J_3(J_1)), J_3(J_1))$ , 要求: ①画出  $J_1$  的有根有序有向图; ②画出  $J_1$  的带头结点的单链表示法。

5-23 设计一个既不用栈，又不修改指针值作标志的算法。

5-24 模仿线索化树的形式，设计一种线索化表，写出对该表的双链表示法进行线索化的算法。

5-25 在图 5.18 (a) 中给出了一个阶为  $4 \times 4$  的方阵, 在图 5.18 (b) 中的每个数字是  $\mathbf{A}$  中与这个数字相对应的元素在顺序存储  $\mathbf{A}$  时的序号。假设  $a_{11}$  地址为  $\text{LOC}[a_{11}]$ , 试写出存储  $\mathbf{A}$  中元素  $a_{ij}$  的地址公式。

5-26 在图 5.19 (a) 中给出一个三角阵  $A$ , 在图 5.19 (b) 中的数字是  $A$  中与这个数字相对应的元素在顺序存储  $A$  时的序号。假设  $a_{1,1}$  的地址为  $LOC(a_{1,1})$ , 试写出存储  $A$  中元素  $a_{i,j}$  的地址公式, 在公式中取  $A$  为一个  $n \times n$  阶矩阵。



(a)

图 5.18 习题 5-25 的图

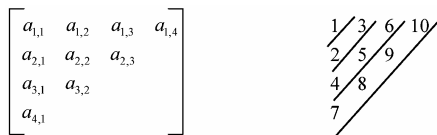


图 5.19 习题 5-26 的图

5-27 设给定的  $n$  维数组  $a[l_1 \cdots u_1, l_2 \cdots u_2, \cdots, l_n \cdots u_n]$ , 如果  $a[l_1, l_2, \cdots, l_n]$  的存储地址为  $\delta$ , 试求  $a[i_1, i_2, \cdots, i_n]$  存储地址。

5-28 已知两个以数组表示的集合  $A$ 、 $B$ 。试设计如下算法：①求并  $A \cup B$ ；②求交  $A \cap B$ ；③求真子集  $A \subset B$ ；④求差  $A - B$ 。



## 第6章 树与二叉树

前面几章介绍的都是线性的数据结构，线性结构虽然具有逻辑关系简单的特点，但对于描述现实世界中一些具有层次（或分支）关系的数据，如人类社会的族谱、各种社会组织机构、博弈过程等，则有很大难度。本章将介绍一种非常重要的非线性结构——树结构。树结构中每个元素最多只有一个前驱，但可能有多个后继，体现出明显的层次关系。本章重点介绍树和二叉树的特点、存储结构，以及各种操作及其相互转换关系等。

### 6.1 树的相关概念

#### 6.1.1 树的递归定义和逻辑表示法

树（Tree）是  $n$  ( $n \geq 0$ ) 个结点的有限集  $T$ ，当  $T$  为空时称为空树，否则它满足以下两个条件：①有且仅有一个特定的数据元素称为根（root）的结点，根结点没有前驱结点；②其余的结点可分为  $m$  ( $m > 0$ ) 个互不相交的子集  $T_1, T_2, \dots, T_m$ ，其中每个集合  $T_i$  ( $1 \leq i \leq m$ ) 本身又是一棵树，并称其为根的子树（subtree）。

树的递归定义刻画了树的固有特性：一棵非空树是由若干棵子树构成的，而子树又可由若干棵更小的子树构成。因此，树结构的很多算法都使用了递归方法。

例如，图 6.1 所示的是一棵具有 10 个结点的树，其中 A 是根，其余结点分成两个互不相交的子集： $T_1 = \{B, D, E, H, I\}$ ， $T_2 = \{C, F, G, J\}$ ； $T_1$  和  $T_2$  都是根 A 的子树，且本身也是一棵树。例如  $T_1$ ，其根为 B，其余结点分为两个互不相交的子集： $T_{11} = \{D\}$ ， $T_{12} = \{E, H, I\}$ 。 $T_{11}$  和  $T_{12}$  都是 B 的子树，而  $T_{12}$  中 E 是根， $\{H\}$  和  $\{I\}$  是两棵互不相交的子树，其本身又是只有一个根结点的树。

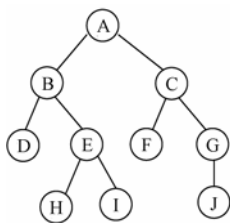


图 6.1 树的示例

#### 6.1.2 树的基本术语

下面给出树结构的一些基本术语。

(1) 结点的度（degree）：结点拥有的子树数。例如，图 6.1 中 A 的度为 2，G 的度为 1，H 的度为 0。

(2) 叶子（终端结点）：度为 0 的结点。图 6.1 中，结点 D、H、I、F、J 都是树的叶子。

(3) 非终端结点：度不为 0 的结点。

(4) 结点的层次：树中根结点的层次为 1，其子树的根为第 2 层，其余类推。

(5) 树的度：树中所有结点的最大度数。例如，如图 6.1 所示的树的度为 2。

(6) 树的深度：树中所有结点层次的最大值。例如，如图 6.1 所示的树深度为 4。

(7) 有序树、无序树：如果树中每棵子树从左向右的排列拥有一定的顺序，不得互换，则称为有序树；否则称为无序树。

(8) 森林： $m$  ( $m \geq 0$ ) 棵互不相交的树的集合。

在树结构中，结点之间的关系又可用家族关系描述，定义如下。

(9) 孩子、双亲：结点子树的根称为这个结点的孩子，而这个结点又被称为孩子的双亲。例如，在如图 6.1 所示的树中，B 为 A 的子树  $T_1$  的根，则 B 是 A 的孩子，而 A 则是 B 的双亲。

(10) 子孙：以某结点为根的子树中的所有结点都被称为是该结点的子孙。图 6.1 中，B 的子孙为 E、D、H 和 I。

(11) 祖先：从根结点到该结点路径上的所有结点。图 6.1 中，H 的祖先为 A、B 和 E。

(12) 兄弟：同一个双亲的孩子之间互为兄弟。图 6.1 中，D 和 E 互为兄弟，F 和 G 互为兄弟。

(13) 堂兄弟：双亲在同一层的结点互为堂兄弟。图 6.1 中，I 和 J 互为堂兄弟。

### 6.1.3 树的抽象类型定义

树的抽象数据类型定义见 ADT 6-1。

ADT 6-1 树的抽象类型定义

```
ADT Tree{
    数据集合 D: D 是具有相同特性的数据元素的集合。
    数据关系 R: 若 D 为空集，则称为空树；若 D 仅含一个数据元素，则 R 为空集，否则
    R={H}, H 是如下二元关系:
        (1) 在 D 中存在唯一的称为根的数据元素 root，它在关系 H 下无前驱；
        (2) 若  $D - \{root\} \neq \emptyset$ ，则存在  $D - \{root\}$  的一个划分  $D_1, D_2, \dots, D_m$  ( $m > 0$ )，对任意
         $j \neq k$  ( $1 \leq j, k \leq m$ ) 有  $D_i \cap D_k = \emptyset$ ，且对任意的  $i$  ( $1 \leq i \leq m$ )，唯一存在数据元素  $x_i \in D_i$ ，
        有  $\langle root, x_i \rangle \in H$ ；
        (3) 对应于  $D - \{root\}$  的划分， $H - \{\langle root, x_1 \rangle, \dots, \langle root, x_m \rangle\}$  有唯一的一个划分  $H_1, H_2, \dots, H_m$ 
        ( $m > 0$ )，对任意  $j \neq k$  ( $1 \leq j, k \leq m$ ) 有  $H_j \cap H_k = \emptyset$ ，对任意  $i$  ( $1 \leq i \leq m$ )， $H_i$  是  $D_i$ 
        上的二元关系， $(D_i, \{H_i\})$  是一棵符合本定义的子树，称为根 root 的子树。
    数据操作:
        Init_Tree(&T);  输出: 空树 T。
        Destroy_Tree(&T);
            输入: 树 T。输出: 删除树 T。
        Create_Tree(&T, definition);
            输入: definition 给出树 T 的定义。输出: 按 definition 构造的树 T。
        Clear_Tree(&T);
            输入: 树 T。输出: 树 T 为空树。
        Tree_Empty(T);
            输入: 树 T。输出: 若 T 为空树，则返回 TRUE；否则返回 FALSE。
        Tree_Depth(T);
            输入: 树 T。输出: T 的深度。
        Root(T);
            输入: 树 T。输出: T 的根。
        Value(T, cur_e);
            输入: 树 T, cur_e 是 T 中某个结点。输出: cur_e 的值。
}
```

```
Assign(T, cur_e, value);
    输入：树 T，cur_e 是 T 中某个结点。输出：结点 cur_e 赋值为 value。
Parent(T, cur_e);
    输入：树 T，cur_e 是 T 中某个结点。
    输出：若 cur_e 是 T 的非根结点，则返回它的双亲；否则函数值为“空”。
Left_Child(T, cur_e);
    输入：树 T，cur_e 是 T 中某个结点。
    输出：若 cur_e 是 T 的非叶子结点，则返回它的最左孩子；否则返回“空”。
Right_Sibling(T, cur_e);
    输入：树 T，cur_e 是 T 中某个结点。
    输出：若 cur_e 有右兄弟，则返回它的右兄弟；否则函数值为“空”。
Insert_Child(&T, &p, i, c);
    输入：树 T，p 指向 T 中某个结点， $1 \leq i \leq p$  所指结点的度+1，非空树 c 与 T 不相交。
    输出：插入 c 作为 T 中 p 所指结点的第 i 棵子树。
Delete_Child(&T, &p, i);
    输入：树 T，p 指向 T 中某个结点， $1 \leq i \leq p$  所指结点的度。
    输出：删除 T 中 p 所指结点的第 i 棵子树。
Traverse_Tree(T, visit());
    输入：树 T，visit 是对结点操作的应用函数。
    输出：按某种次序对 T 的每个结点调用函数 visit() 一次且至多一次。一旦 visit()
    失败，则操作失败。
}ADT Tree
```

---

## 6.2 树的存储结构与遍历

在现实应用中，数据之间的关系更多地表现为树结构。下面将讨论树的表示及其遍历操作。

### 6.2.1 树的存储结构

#### 1. 双亲表示法

由树的定义可知，树中任意一个结点的双亲只有一个，因此，在用一组连续空间存储树中结点的信息时，在每个结点中附设一个指向其双亲的指针来指示其双亲结点在链表中的位置。这种方式就是双亲表示法，其形式说明见程序 6-1。该类成员函数没有完整给出，留作练习。

程序 6-1 树的双亲表示法存储表示

---

```
template<class T>
class PTNode { // 结点结构
public:
    T data;
    int parent; // 双亲位置域
};
```

```
class PTree { //树结构
    PNode    *nodes;
    int  r, n; //根的位置和结点数
    int  MaxSize;
public:
    PTree(int MaxTreeSize); //PTree 类构造函数
    //...
};
```

图 6.2 显示了一棵树及其双亲表示的存储结构。取某结点的双亲结点的操作 PARENT(T, x)可在常量时间内实现。反复调用 PARENT 操作，直到遇见无双亲的结点为止，便找到了树的根，这就是 ROOT(x)操作的执行过程。但是，在这种表示法中，求结点的孩子则需要遍历整个结构。

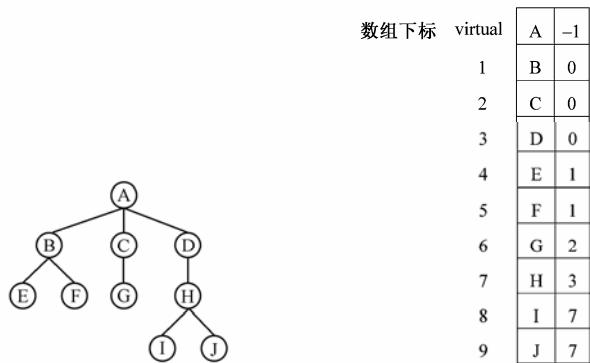


图 6.2 树的双亲表示法示例

2. 孩子表示法

根据树的特点，它的每个结点都可能子结点，且子结点的个数不确定，因此可给每个结点设置多个指针域，每个指针指向一棵子树的根结点，此方式称为孩子表示法。该表示法的存储方法之一是，采用多重链表的方式，即根据树的度  $d$  为每个结点设置  $d$  个指针域，如图 6.3 (a) 所示的结点格式。

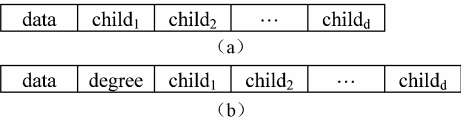


图 6.3 结点格式

显然，这种多重链表中的结点是同构的。采用这种结构时，一棵具有  $n$  个结点、度为  $d$  的树中共有  $nd$  个指针域，由于树中只有  $n-1$  个分支，也就只用到  $n-1$  个指针域，因此必有  $nd-(n-1) = n(d-1)+1$  个空链域，空间浪费较大。若采用图 6.3 (b) 所示的结点格式，即链表中的结点具有不同结构，其中  $d$  为结点的度，degree 域的值同  $d$ ，此时，虽能节约存储空间，但操作不方便。

另一种方法是把每个结点的孩子结点排列起来，形成一个单链表，则  $n$  个结点就有  $n$  个链表，其中，叶子的孩子链表为空表，链表中增加一个头结点，为方便查找，将这些头结点采用顺序存储结构。图 6.4 (a) 所示是图 6.2 中树的孩子表示法。与双亲表示法相反，孩子表示法便于实现对孩子进行的操作，但却不适合对双亲结点进行的操作，为此，可把双亲

表示法和孩子表示法结合起来，形成另一个表示方法，如图 6.4（b）所示。

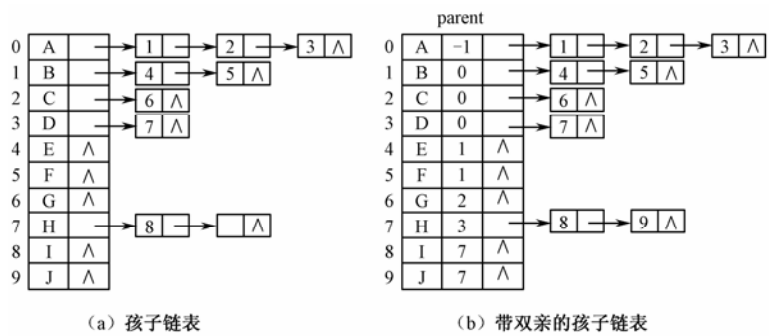


图 6.4 图 6.2 的树的另外两种表示法

程序 6-2 给出孩子表示法的链存储结构的 C++语言描述，但不完整，完整的定义留作练习。

程序 6-2 树的孩子表示法链表存储表示

```
class CTNode{//孩子结点
    int nodeId;
    CTNode *next;
public: //...
};

template<class T>
class CTBox{
    T data; //树元素内容
    CTNode *firstSon; //链头指针，实质上指向它与它的第一个儿子对应的边
    int fatherId; //父亲索引，即父亲元素结点在数组中的位置
    int sonNum; //儿子数目
public: //...
};

template<class T>
class CCTree {
    CTBox *tree;
    int MaxSize, r; //结点数和根的位置
public:
    CCTree(int maxNodeSize);
    virtual ~CCTree();
};
```

3. 孩子兄弟表示法

此方法以二叉链表作为树的存储结构。与后面所述的二叉树的链表表示方法不同的是，这里的链表结点的两个链域分别指向该结点的第一个孩子结点和下一个兄弟结点，称为 firstchild 域和 nextsibling 域。图 6.5 是图 6.2 中树的孩子兄弟

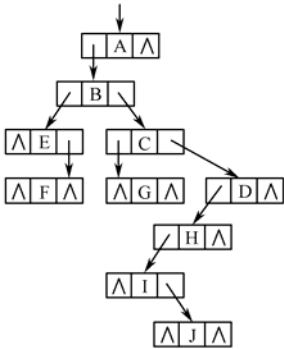


图 6.5 图 6.2 中树的二叉链表

链表。

这种存储方式最大的优点在于能很方便地利用后面所述的二叉树的算法来实现树的各种操作。首先，易于实现找结点孩子的操作。其次，如果给每个结点再加上双亲域，也能很方便地找到结点的双亲。程序 6-3 给出该方法的存储描述。

程序 6-3 树的二叉链表（孩子—兄弟）存储表示

```
template<class T>
class CSNode
{
    T data;
    CSNode *firstchild, *nextsibling; //大儿子指针和下个兄弟指针
public:
    CSNode<T> *GetFirstChild( ) { return firstchild; };
    CSNode<T> *GetNextSibling( ) { return nextsibling; };
    void SetFirstChild(CSNode<T> *p) { firstchild=p; };
    void SetNextSibling(CSNode<T> *p) { nextsibling=p; };
    void SetData(T &d) { data=d; };
    T &GetData( ) { return data; };
};
```

### 6.2.2 树与森林的遍历

#### 1. 树的遍历

树的遍历通常有以下两种方法。① 先根遍历：先访问树的根结点，然后按照从左到右的顺序依次先根遍历根结点的每一棵子树。② 后根遍历：先按照从左到右的顺序依次后根遍历根结点的每一棵子树，然后访问根结点。

例如，对如图 6.6 所示的树进行遍历，可得树的先根遍历序列为 1 2 3 4 5；若对此树进行后根遍历，则得树的后根序列为 2 5 3 4 1。

#### 2. 森林的遍历

根据森林和树相互递归的定义，可推出森林的两种遍历方法。

（1）先序遍历森林：① 访问森林中第一棵树的根结点；② 先序遍历第一棵树中根结点的子树森林；③ 先序遍历除去第一棵树之后剩余的树构成的森林。

（2）后序遍历森林：① 后序遍历森林中第一棵树的根结点的子树森林；② 访问第一棵树的根结点；③ 后序遍历除去第一棵树之后剩余的树构成的森林。

例如，对图 6.7 中的森林进行先序遍历和后序遍历，分别得到森林的先序序列为 1 2 3 4 5 6 7 8 9 10 11 12，后序序列为 5 2 3 4 8 9 7 6 12 11 10。

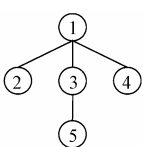


图 6.6 一棵树

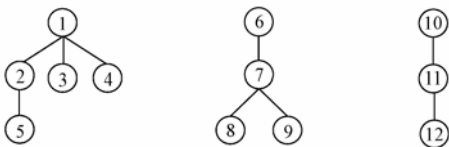


图 6.7 由 3 棵树构成的森林

**【例 6-1】** 试用 C++语言编写按层次遍历一棵给定的  $m$  次树的算法。

本题采用标准形式存储给定的  $m$  次树，同时使用一个队列 `queue` 来存放还没有处理的子树的根结点的地址。算法如下：

```
const int n=500; //树中结点的个数
const int m=5; //树的次数
const int l=10; //结点中字符的个数
struct treenode
{   char data[l]; int child[m];
};
treenode node[n]; //node 用于存放树结点
int root; //root 指向根结点
void levorder(int t)
{   int queue[n]; //队列 queue
    int head, tail, p, i; //head, tail 分别是 queue 队列的头、尾指针
    queue[0]=t; head=0; tail=0;
    while (head<=tail)
    {   p=queue[head]; head++; cout<<node[p].data<<endl;
        for (i=0; i<m; i++)
        {   if (node[p].child[i]!=0) { tail++; queue[tail]=node[p].child[i]; }}
    }
```

## 6.3 二 叉 树

二叉树是树结构的一个重要类型，许多实际问题抽象出来的数据结构都是二叉树的形式，一般的树也能简单地转换为二叉树，而且二叉树的存储结构及其算法都较为简单，因此二叉树特别重要。

### 6.3.1 二叉树的定义

一棵二叉树（binary tree）是结点的一个有限集合，该集合或者为空，或者由一个根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。二叉树的抽象数据类型定义见 ADT 6-2。

ADT 6-2 二叉树的抽象类型定义

ADT BinaryTree{

数据集合  $D$ :  $D$  是具有相同特性的数据元素的集合。

数据关系  $R$ : 若  $D=\emptyset$ ，则  $R=\emptyset$ ，称 BinaryTree 为空二叉树；若  $D!\!=\emptyset$ ，则  $R=\{H\}$ ， $H$  是如下二元关系：

- (1) 在  $D$  中存在唯一的称为根的数据元素  $root$ ，它在关系  $H$  下无前驱；
- (2) 若  $D - \{root\} \neq \emptyset$ ，则存在  $D - \{root\} = \{D_l, D_r\}$ ，且  $D_l \cap D_r = \emptyset$ ；
- (3) 若  $D_l \neq \emptyset$ ，则  $D_l$  中存在唯一的元素  $x_l$ ， $\langle root, x_l \rangle \in H$ ，且存在  $D_l$  上的关系  $H_l$  为  $H$  的子集；若  $D_r \neq \emptyset$ ，则  $D_r$  中存在唯一的元素  $x_r$ ， $\langle root, x_r \rangle \in H$ ，且存在  $D_r$  上的关系  $H_r$  为  $H$  的子集。

子集:  $H = \{\langle \text{root}, x_l \rangle, \langle \text{root}, x_r \rangle, H_l, H_r\}$ ;

(4)  $(D_l, \{H_l\})$  是一棵符合本定义的二叉树, 称为根的左子树,  $(D_r, \{H_r\})$  是一棵符合本定义的二叉树, 称为根的右子树。

数据操作:

Init\_BiTree(&T); //初始化二叉树

输出: 空二叉树 T。

Destroy\_BiTree(&T); //删除二叉树

输入: 二叉树 T。输出: 删除二叉树 T。

Create\_BiTree(&T, definition); //创建二叉树

输入: definition 给出二叉树 T 的定义。输出: 按 definition 构造二叉树 T。

Clear\_BiTree(&T); //清空二叉树

输入: 二叉树 T。输出: 二叉树 T 清为空树。

If\_Empty\_BiTree (T); //判断二叉树是否为空

输入: 二叉树 T。输出: 若 T 为空二叉树, 则返回 TRUE; 否则返回 FALSE。

Depth\_of\_BiTree (T); //求二叉树的深度

输入: 二叉树 T。输出: T 的深度。

BiTree\_Root(T); //得到二叉树的根结点

输入: 二叉树 T。输出: T 的根。

BiTree\_Value(T, e); //取二叉树某个结点的值

输入: 二叉树 T, e 是 T 中某个结点。输出: e 的值。

BiTree\_Assign(T, &e, value); //给二叉树的某个结点赋值

输入: 二叉树 T, e 是 T 中某个结点。输出: 结点 e 赋值为 value。

BiTree\_Parent(T, e); //求二叉树某个结点的双亲结点

输入: 二叉树 T, e 是 T 中某个结点。

输出: 若 e 是 T 的非根结点, 则返回它的双亲; 否则函数值为“空”。

BiTree\_Left\_Child(T, e); //取二叉树某个结点的左孩子

输入: 二叉树 T, e 是 T 中某个结点。

输出: 返回 e 的左孩子。若 e 无左孩子, 则返回“空”。

BiTree\_Right\_Child(T, e); //取二叉树某个结点的右孩子

输入: 二叉树 T, e 是 T 中某个结点。

输出: 返回 e 的右孩子。若 e 无右孩子, 则返回“空”。

BiTree\_Left\_Sibling(T, e); //取二叉树某个结点的左兄弟

输入: 二叉树 T, e 是 T 中某个结点。

输出: 返回 e 的左兄弟。若 e 是 T 的左孩子或无左兄弟, 则返回“空”。

BiTree\_Right\_Sibling(T, e); //取二叉树某个结点的右兄弟

输入: 二叉树 T, e 是 T 中某个结点。

输出: 返回 e 的右兄弟。若 e 是 T 的右孩子或无右兄弟, 则返回“空”。

BiTree\_Insert\_Child(T, P, LR, c); //插入结点

输入: 二叉树 T, p 指向 T 中某个结点, LR 为 0 或 1, 非空二叉树 c 与 T 不相交且右子树为空。

输出: 根据 LR 为 0 或 1, 插入 c 为 T 中 p 指向结点的左或右子树。p 所指结点的原有左或右子树成为 c 的右子树。

BiTree\_Delete\_Child(T, P, LR); //删除结点



输入：二叉树 T，p 指向 T 中某个结点，LR 为 0 或 1。

输出：根据 LR 为 0 或 1，删除 T 中 p 所指结点的左或右子树。

BiTree\_PreOrder\_Traverse(T, visit()); //按先序遍历二叉树

输入：二叉树 T，visit 是对结点操作的应用函数。

输出：先序遍历 T，对每个结点调用函数 visit() 一次且仅一次。一旦 visit() 失败，则操作失败。

BiTree\_InOrder\_Traverse(T, visit()); //按中序遍历二叉树

输入：二叉树 T，visit 是对结点操作的应用函数。

输出：中序遍历 T，对每个结点调用函数 visit() 一次且仅一次。一旦 visit() 失败，则操作失败。

BiTree\_PostOrder\_Traverse(T, visit()); //按后序遍历二叉树

输入：二叉树 T，visit 是对结点操作的应用函数。

输出：后序遍历 T，对每个结点调用函数 visit() 一次且仅一次。一旦 visit() 失败，则操作失败。

BiTree\_LevelOrder\_Traverse(T, visit()); //按层序遍历二叉树

输入：二叉树 T，visit 是对结点操作的应用函数。

输出：层序遍历 T，对每个结点调用函数 visit() 一次且仅一次。一旦 visit() 失败，则操作失败。

}ADT BinaryTree

二叉树的 5 种基本形态，如图 6.8 所示：

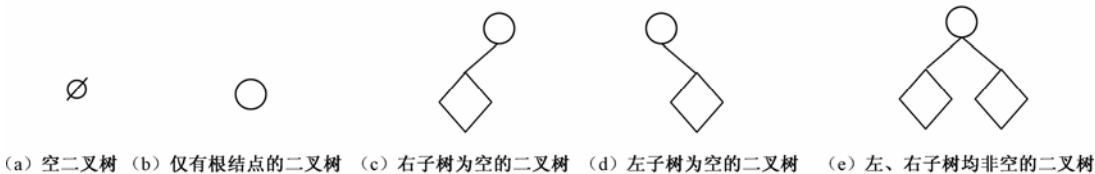


图 6.8 二叉树的 5 种基本形态

### 6.3.2 二叉树的性质

二叉树具有下列重要特性。

**性质 1** 在二叉树的第  $i$  层上至多有  $2^{i-1}$  个结点 ( $i \geq 1$ )。

请读者自己用数学归纳法证明。

**性质 2** 深度为  $k$  的二叉树至多有  $2^k - 1$  个结点 ( $k \geq 1$ )。

请读者利用性质 1 证明性质 2。

**性质 3** 对任意一棵非空二叉树 T，若其叶结点数为  $n_0$ ，度为 2 的结点数为  $n_2$ ，则  $n_0 = n_2 + 1$ 。

请读者利用二叉树的定义（所有结点的度均小于或等于 2）和计算分支数等证明性质 3。

下面介绍两种特殊的二叉树：完全二叉树和满二叉树。满二叉树（full binary tree）是指深度为  $k$  且有  $2^k - 1$  个结点的二叉树。图 6.9 (a) 所示是深度为 4 的满二叉树，这种树的特点是，每一层上结点数都是最大结点数，即不存在度为 1 的结点。

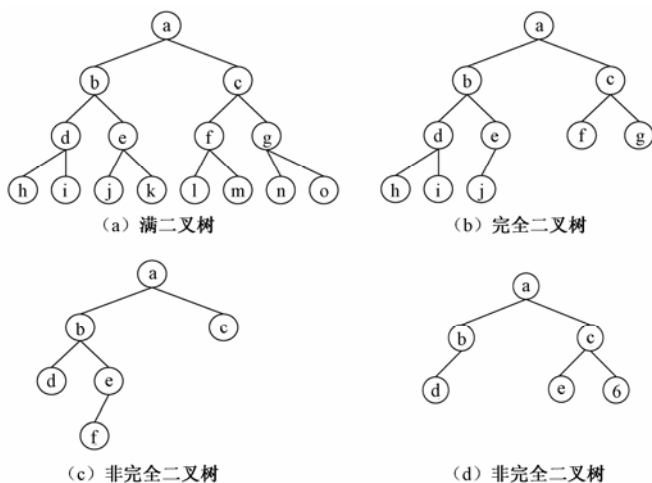


图 6.9 特殊形态的二叉树

完全二叉树 (completed binary tree) 是指深度为  $k$ , 有  $n$  个结点的二叉树, 除最后一层外, 其余层均是满的, 且最下面一层的结点都集中在最左边的位置上。如图 6.9 (b) 所示为一棵深度为 4 的完全二叉树, 而图 6.9 (c) 和图 6.9 (d) 所示都不是完全二叉树。下面介绍完全二叉树的两个重要特性。

**性质 4** 具有  $n$  个结点的完全二叉树的深度为  $\lfloor \log_2 n \rfloor + 1$ 。

证明: 设完全二叉树的深度为  $k$ , 由它的定义和性质可知  $2^{k-1} - 1 < n \leq 2^k - 1$  或  $2^{k-1} \leq n < 2^k$ , 取对数后有  $k-1 \leq \log_2 n < k$ , 因为  $k$  是整数, 所以  $k = \lfloor \log_2 n \rfloor + 1$ 。

**性质 5** 如果将一棵有  $n$  个结点的完全二叉树 (其深度为  $\lfloor \log_2 n \rfloor + 1$ ) 自顶向下、同一层自左向右编号为  $1, 2, 3, \dots, n$ , 则对任意一个结点  $i$  ( $1 \leq i \leq n$ ) 有: ① 如果  $i = 1$ , 则此结点为二叉树的根, 无双亲; 如果  $i > 1$ , 则其双亲结点是  $\text{floor}(i/2)$ 。② 如果  $2i > n$ , 则结点  $i$  无左孩子 (结点  $i$  为叶子结点); 否则其左孩子是结点  $2i$ 。③ 如果  $2i+1 > n$ , 则结点  $i$  无右孩子; 否则其右孩子是结点  $2i+1$ 。

请读者先证明性质 5 的②和③, 再从②和③推导出①。

## 6.4 二叉树的存储结构

二叉树通常采用两种存储方式: 顺序存储结构和链式存储结构。

### 6.4.1 顺序存储结构

对于完全二叉树, 可采用一组地址连续的存储单元按照自上而下、从左至右的顺序存储树上的结点元素, 即编号为  $i$  的结点元素存储在一维数组的下标为  $i-1$  的分量中。例如, 图 6.10 (a) 所示为图 6.9 (b) 所示完全二叉树的顺序存储结构。但对于一般二叉树, 为了使结点的存储位置能反映出结点之间的逻辑关系, 应将其每个结点与完全二叉树上的结点相对应, 存储在一维数组的相应分量中。图 6.9 (c) 所示二叉树的顺序存储结构如图 6.10 (b) 所示, 图中以 “0” 表示不存在此结点。显然, 这样会造成空间的浪费。

可见，这种顺序存储结构仅适用于完全二叉树。因为在最坏情况下，一个深度为  $k$  且只有  $k$  个结点的单枝树（树中不存在度为 2 的结点）却需要长度为  $2^k-1$  的一维数组，如图 6.11 所示。

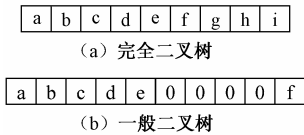


图 6.10 二叉树的顺序存储结构

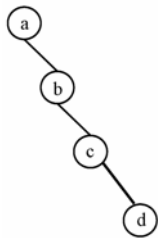


图 6.11 只有  $k$  个结点的单枝树

6.4.2 链式存储结构

二叉树一般多采用链式存储，链结点除存放与树结点有关的信息外，还可根据具体应用的需要，设置分别指向双亲结点，左、右子结点的指针。根据指针设置情况，存储方式可分为一指针式、二指针式和三指针式。

(1) 一指针式

该方法也称父指针式，即每个结点只设立指向双亲结点的指针，在这种存储方式下，只有从每个叶子出发，才能访问到一棵树中的每个结点。如果已知某个结点的指针，可方便地找到其双亲结点，但找它的子结点很耗时。一指针式的结点结构如图 6.12 (a) 所示，存储示例如图 6.13 (b) 所示。

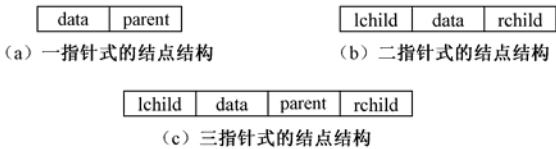


图 6.12 二叉树的结点存储结构

(2) 二指针式

该方法为每个结点只设立指向其后继的指针，分别称为左指针和右指针。在这种存储方式下，从根出发可访问到所有结点，并已知某结点的指针，很容易找到它的子结点，但要找到它的双亲结点，则需从根结点开始搜索，很耗时。二指针式的结点结构如图 6.12 (b) 所示，存储示例如图 6.13 (c) 所示。

容易证明，在含有  $n$  个结点的二叉链表中有  $n+1$  个空链域。在下面章节中将会看到，可利用这些空链域存储其他有用信息，从而得到另一种链式存储结构——线索链表。

(3) 三指针式

该方法为每个结点分别设立指向其前驱和两个后继的三个指针，这种方式同时具有一指针式和二指针式的优点，当然是通过存储空间换来的。三指针式的结点结构如图 6.12 (c) 所示，存储示例如图 6.13 (d) 所示。

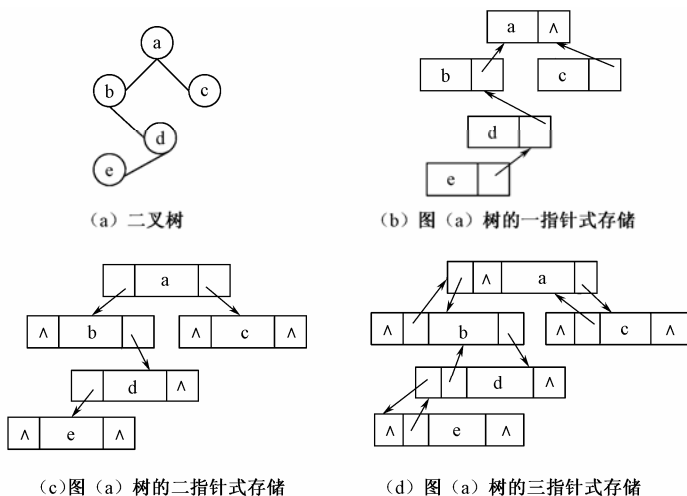


图 6.13 二叉树链式存储结构示例

## 6.5 二叉树对象模型

### 6.5.1 二叉树结点对象

#### (1) 结点的抽象类

结点类 `TBinTreeNode` 定义了结点本身的基本操作接口（见程序 6-4）。这里采用了最小化原则，即只设置针对结点本身的内容和关系的操作，对涉及全局的操作，将放到“树类”中。

程序 6-4 二叉树二指针式结点抽象类

```
class TBinTreeNode
{
protected:
    T data; //树结点内容，类型为可变类型（类模板）
public:
    TBinTreeNode( ) { };
    //构造函数，构造一个数据域为 d 的结点
    virtual ~TBinTreeNode( ) { };
    virtual T& GetData( ) {return data; }; //读元素内容
    virtual TBinTreeNode<T> *SetData(T &e) { data=e; return this; };
    //置元素值
    virtual TBinTreeNode<T> *GetLeftChild( )=0; //读元素的左儿子指针
    virtual TBinTreeNode<T> *GetRightChild( )=0; //读元素的右儿子指针
    virtual void SetLeftChild(TBinTreeNode<T> *pChild)=0; //置左儿子指针
    virtual void SetRightChild(TBinTreeNode<T> *pChild)=0; //置右儿子指针
    int IsLeaf( ) //判断是否为叶子
    { return GetLeftChild( )==NULL && GetRightChild( )==NULL; };
};
```

```
};
```

## (2) 指针式结点类

从上面给出的抽象类 `TBinTreeNode`，可派生出各种具体存储结构的二叉树结点。下面给出一种动态的实现方式：二指针式链结构 `TBinTreeNode`，见程序 6-5。它适合使用堆空间（通过 C++ 语言的 `new`、`malloc` 等建立）实现。

程序 6-5 二叉树二指针式结点抽象类

```
#include "TBinTreeNode.h"
template<class T>
class BinTreeNode:public TBinTreeNode<T>
{
protected:
    BinTreeNode<T> *lc, *rc; //左儿子指针、右儿子指针
public:
    BinTreeNode( ):lc(NULL), rc(NULL) {};
    BinTreeNode(Td, BinTreeNode<T> *lp=NULL, BinTreeNode<T> *rp=NULL):
        data(d), lc(lp), rc(rp) {};
    //构造函数，构造一个数据域为 d 的结点
    virtual ~BinTreeNode( ) {};
    virtual TBinTreeNode<T> *GetLeftChild( ) { return lc; };
    //读元素的左儿子指针
    virtual TBinTreeNode<T> *GetRightChild( ) { return rc; };
    //读元素的右儿子指针
    virtual void SetLeftChild(TBinTreeNode<T> *pChild)
    { lc=(BinTreeNode<T> *)pChild; }; //置左儿子指针
    virtual void SetRightChild(TBinTreeNode<T> *pChild)
    { rc=(BinTreeNode<T> *)pChild; }; //置右儿子指针
};
```

## 6.5.2 二叉树对象

二叉树类 `TBinTree` 定义了一个二叉树实体及其相关操作，该实体由一个类型为 `TBinTreeNode` 的根指针 `root` 指向。由于在 C++ 语言中，指向基类的指针可直接指向该基类的派生类，并在树结点类中通过基本操作抽象隐蔽了结点的结构，因此该二叉树类兼容各种具体的树结点，程序 6-6 是该类的 C++ 语言描述。

程序 6-6 二叉树类定义

```
#define NULL 0
#include "BinTreeNode.h"
#include "SeqStack.h"
#include "LinkedQueue.h"
#include "exception.h"
#include<iostream.h>
```

```

template<class T>
class BinaryTree
{
public :
    BinaryTree( ) { root=0; };
    BinaryTree(TBinTreeNode<T> *p) { root=p; };
    BinaryTree(Td) { root=new TBinTreeNode<T>; root->SetData(d); };
    virtual ~BinaryTree( ) { Destroy(root); };
    virtual void CreateBinaryTree( ) { root=CreateBinTree( ); };
    virtual TBinTreeNode<T> *CreateBinTree( );
    virtual bool IsEmpty( ) const{return((root)? false:true); }
    virtual TBinTreeNode<T> *Root( ) const{ return root; }; //返回树根
    virtual TBinTreeNode<T> *Parent(TBinTreeNode<T> *p) { return Parent(root, p); }
    //找 p 的双亲
    TBinTreeNode<T> *LeftSibling(TBinTreeNode<T> *p); //找 p 的左兄弟
    TBinTreeNode<T> *RightSibling(TBinTreeNode<T> *p); //找 p 的右兄弟
    bool InsertLeftChild(TBinTreeNode<T> *p, T d);
    bool InsertRightChild(TBinTreeNode<T> *p, T d);
    bool DeleteLeftChild(TBinTreeNode<T> *p);
    bool DeleteRightChild(TBinTreeNode<T> *p);
    virtual void PreOrder( ) { PreOrder(root); };
    virtual void InOrder( ) { InOrder(root); };
    virtual void PostOrder( ) { PostOrder(root); };
    virtual void LevelOrder( );
    virtual void InOrderTraverse( ) { InOrderTraverse(root); };
    virtual void InOrderTraverse2( ) { InOrderTraverse2(root); };
protected:
    TBinTreeNode<T> *root; //根结点指针
    TBinTreeNode<T> *Parent(TBinTreeNode<T> *start, TBinTreeNode<T> *p);
    //从 start 开始, 查找 p 的双亲
    void Destroy(TBinTreeNode<T> *p); //删除以 p 为根的二叉树
    void PreOrder(TBinTreeNode<T> *r); //前序遍历以 r 为根的二叉树
    void InOrder(TBinTreeNode<T> *p); //中序遍历以 r 为根的二叉树
    void PostOrder(TBinTreeNode<T> *p); //后序遍历以 r 为根的二叉树
    virtual void InOrderTraverse(TBinTreeNode<T> *r);
    virtual void InOrderTraverse2(TBinTreeNode<T> *r);
};

```

---

程序 6-7 给出部分成员函数的实现, 二叉树遍历的算法将在 6.6 节中介绍。

#### 程序 6-7 二叉树部分成员函数的实现

---

```

template<class T>
void BinaryTree<T>::Destroy(TBinTreeNode<T> *p) //删除以 p 为根的二叉树
{
    if (p!=NULL)

```

```

        { Destroy(p->GetLeftChild( )); Destroy(p->GetRightChild( ));
          delete p;
        }
    }
}

template<class T>
TBinTreeNode<T> *BinaryTree<T>::
Parent(TBinTreeNode<T> *r, TBinTreeNode<T> *p) //找 p 的双亲
{
    TBinTreeNode<T> *q;
    if (r==NULL) return NULL;
    if (r->GetLeftChild( )==p || r->GetRightChild( )==p) return r;
    if (q=Parent(r->GetLeftChild( ), p))!=NULL) return q;
    else return Parent(r->GetRightChild( ), p);
}

template<class T>
TBinTreeNode<T> *BinaryTree<T>::LeftSibling(TBinTreeNode<T> *p)
{//找 p 的左兄弟
    TBinTreeNode<T> *q; q=Parent(root, p);
    if ((q==NULL) || (p==q->GetLeftChild( ))) return NULL;
    else return q->GetLeftChild( );
}

template<class T>
TBinTreeNode<T> *BinaryTree<T>::RightSibling(TBinTreeNode<T> *p)
{//找 p 的右兄弟
    TBinTreeNode<T> *q; q=Parent(root, p);
    if ((q==NULL) || (p==q->GetRightChild( ))) return NULL;
    else return q->GetRightChild( );
}

template<class T>
bool BinaryTree<T>::InsertLeftChild(TBinTreeNode<T> *p, T d)
{//为 p 插入值为 d 的左儿子
    if (p==NULL) return false;
    TBinTreeNode<T> *q=new TBinTreeNode<T>(d);
    q->SetLeftChild(p->GetLeftChild( )); p->SetLeftChild(q);
    return true;
}

template<class T>
bool BinaryTree<T>::InsertRightChild(TBinTreeNode<T> *p, T d)
{//为 p 插入值为 d 的右儿子
    if (p==NULL) return false;
    TBinTreeNode<T> *q=new TBinTreeNode<T>(d);
    q->SetRightChild(p->GetRightChild( )); p->SetRightChild(q);
    return true;
}

template<class T>

```

```

bool BinaryTree<T>::DeleteLeftChild(TBinTreeNode<T> *p)
{//删除 p 左儿子
    if (p==NULL) return false;
    Destroy(p->GetLeftChild( ));
    p->SetLeftChild(0); //删除后左儿子指针指空
    return true;
}

template<class T>
bool BinaryTree<T>::DeleteRightChild(TBinTreeNode<T> *p)
{//删除 p 右儿子
    if (p==NULL) return false;
    Destroy(p->GetRightChild( )); //删除后右儿子指针指空
    p->SetRightChild(0);
    return true;
}

```

---

## 6.6 二叉树的遍历与线索化

### 6.6.1 二叉树的遍历

#### 1. 定义

所谓遍历二叉树（traversing binary tree）就是按某种顺序访问二叉树中的每个结点一次且仅一次的过程。这里的“访问”可以是输出、比较、更新、查看元素内容等各种操作。

#### 2. 遍历递归算法

由前面给出的二叉树递归定义可知，二叉树是由根结点、左子树和右子树三个基本单元组成的，设 L、D、R 分别表示这三个部分，根据对这三个部分遍历次序的不同，可有 DLR、LDR、LRD、DRL、RDL 和 RLD 共 6 种遍历方案。若限定先左后右，则只有前三种情况，分别称为先（根）序遍历、中（根）序遍历和后（根）序遍历。基于二叉树的递归定义，可得下述遍历二叉树的递归算法定义。

##### （1）先序遍历二叉树

若二叉树为空，则空操作，否则，依次执行如下操作：① 访问根结点；② 先序遍历左子树；③ 先序遍历右子树。

##### （2）中序遍历二叉树

若二叉树为空，则空操作，否则，依次执行如下操作：① 中序遍历左子树；② 访问根结点；③ 中序遍历右子树。

##### （3）后序遍历二叉树

若二叉树为空，则空操作，否则，依次执行如下操作：① 后序遍历左子树；② 后序遍历右子树；③ 访问根结点。

假设二叉树的存储结构为二叉链表，则遍历二叉树的递归实现过程见程序 6-8。



```

template<class T>
void BinaryTree<T>::PreOrder(TBinTreeNode<T> *r)
{
    if (r!=NULL)
    {
        cout<<r->GetData( )<<endl;
        PreOrder(r->GetLeftChild( ));
        PreOrder(r->GetRightChild( ));
    }
}

template<class T>
void BinaryTree<T>::InOrder(TBinTreeNode<T> *r)
{
    if (r!=NULL)
    {
        InOrder(r->GetLeftChild( ));  cout<<r->GetData( )<<endl;
        InOrder(r->GetRightChild( ));
    }
}

template<class T>
void BinaryTree<T>::PostOrder(TBinTreeNode<T> *r)
{
    if (r!=NULL)
    {
        PostOrder(r->GetLeftChild( ));
        PostOrder(r->GetRightChild( ));  cout<<r->GetData( )<<endl;
    }
}

```

下面以图 6.14 所示的二叉树为例进行说明。先序遍历此二叉树，按访问结点的先后次序将结点排列起来，可得二叉树的先序序列为 **a b c d e f g h i**；类似地，中序遍历此二叉树，可得二叉树的中序序列为 **d b g e a c h f i**；后序遍历此二叉树，可得二叉树的后序序列为 **d g e b h i f c a**。

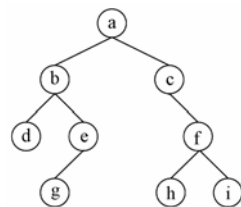


图 6.14 二叉树的遍历

### 3. 遍历非递归算法

采用递归算法实现遍历的优点是可读性好、易于证明、便于程序的编写和调试，但它也具有耗费计算机资源（时间和空间）较多、运行效率低的不足，因此可考虑将递归算法转化为非递归算法。下面以中序遍历为例说明二叉树遍历的非递归算法，该算法可根据递归算法执行过程中递归工作栈的状态变化得出。

设  $S$  为一个工作栈， $p$  为指向根结点的指针，执行过程如下：① 当  $p$  为非空时，将  $p$  指向结点的地址进栈，并将  $p$  指向该结点的左子树；② 当  $p$  为空时，弹出栈顶元素，显示结点元素，并将  $p$  指向该结点的右子树；③ 重复步骤①、②，直到栈为空且  $p$  也为空为止。中序遍历二叉树的非递归算法见程序 6-9。

```
template<class T>
void BinaryTree<T>::InOrderTraverse(BinTreeNode<T> *r)
{ template<class T>
void BinaryTree<T>::InOrderTraverse2(BinTreeNode<T> *r)
{//中序遍历二叉树的非递归算法, 采用二叉链表存储结构
    SeqStack<BinTreeNode<T> *> s; BinTreeNode<T> *p;
    p=r;
    while (p|| !s.IsEmpty())
    {   if (p)   {   s.Push(p); p=p->GetLeftChild(); } //if
        else {   s.Pop(p);
                if (p==NULL) return; else cout<<p->GetData()<<endl;
                p=p->GetRightChild();
            } //else
        } //while
    }
```

**【例 6-2】** 试编写用非递归过程实现二叉树的遍历算法。

用递归过程可简单方便地实现二叉树的前序、中序、后序遍历。但为了克服其低效性, 也可用非递归过程实现二叉树的遍历。现以中序遍历为例, 用栈模拟递归过程, 算法如下:

```
const int stacksize=50;
void inorder(btreenode *p) //用一个栈实现二叉树的中序遍历
{   btreenode *ps[stacksize];
    int top=-1; //栈量空
    do
    {   while(p!= NULL)
        {   top++;
            if (top>stacksize-1) {   cerr<<“栈满!”<<endl; exit(1); }
            ps[top]=p; p=p->child; }
        if (top>=0) { p=ps[top]; top--; cout<<p->data<<endl; p=p->rchild; }
    }
    while (top>=0);
}
```

**【例 6-3】** 试编写算法实现下述运算: ① 为一棵给定的二叉树生成一份准确的副本; ② 判断两棵给定的树是否等价; ③ 判断一棵给定的二叉树是否是为完全二叉树。算法如下:

```
struct btreenode
{   btreenode *lchild, * rchild;
    char data;
```

```

} btreenode *root1, *root2; //root1,root2 分别指向两棵二叉树的根结点

btreenode *copy(btreeode *t) //为一棵给定的二叉树生成一份准确的副本
{
    btreenode *p;
    if (t!=NULL)
    {
        p=new btreenode; p->lchild=copy(t->lchild);
        p->rchild=copy(t->rchild); p->data=t->data; return p;
    }
    else return NULL;
}

//执行下面的调用语句，产生给定二叉树的一份准确副本 root2=copy(root1);
bool equal(btreeode *t1, btreenode *t2)
{
    //判断两棵给定的二叉树是否等价，若等价，则返回 true
    bool x=false;
    if ((t1==NULL)&&(t2==NULL)) x=true;
    else if ((t1!= NULL)&&(t2!= NULL))
    {
        if ((t1->data==t2->data)&&(equal(t1->lchild, t2->lchild))
            &&(equal(t1->rchild, t2->rchild))) x=true;
    }
    return x;
}

bool full(btreeode *t) //判断给定的一棵二叉树是否为完全二叉树
{
    bool x;
    if ((t->lchild=NULL)&&(t->rchild==NULL)) x=true;
    else if ((t->lchild=NULL)|| (t->rchild==NULL)) x=false;
    else x=(full(t->lchild)&& full(t->rchild));
}

```

## 4. 层序遍历

所谓层序遍历（level order traversal）二叉树，是指从二叉树的第一层（根结点）开始，自上至下遍历，在同一层中，按从左到右的顺序逐个访问结点。在进行层序遍历时，需要设置一个队列结构，并按下述步骤层序遍历二叉树。

- ① 初始化队列，并将根结点入队。
- ② 当队列非空时，取出队头结点  $p$ ，转步骤③；如果队列为空，则遍历结束。
- ③ 访问结点  $p$ 。如果该结点有左孩子，则将其左孩子入队列；如果该结点有右孩子，则将其右孩子入队列。
- ④ 重复步骤②、③，直到队列为空为止。

程序 6-10 给出层序遍历算法的 C++语言描述，其中的二叉树存储结构仍为二叉链表，另外还用到了前面定义的队列 `LinkedQueue`。

---

```

template<class T>
void BinaryTree<T>::LevelOrder( )
{
    LinkQueue<TBinTreeNode<T> *> q;
    TBinTreeNode<T> *p=root;
    if (p) q.Insert(p);
    while (!q.IsEmpty( ))
    {
        q.Delete(p); cout<<p->GetData( )<<endl;
        if (p->GetLeftChild( )) q.Insert(p->GetLeftChild( ));
        if (p->GetRightChild( )) q.Insert(p->GetRightChild( ));
    }
}

```

---

## 5. 二叉树遍历的应用——根据先序序列创建树

遍历是对二叉树进行各种操作的基础，可通过遍历实现输出结点信息，获得结点的双亲结点、子结点，判定结点所在层次等操作。下面介绍在遍历过程中生成结点并建立二叉树的存储结构的算法。程序 6-11 给出了按先序序列建立二叉树的二叉链表的算法。对如图 6.13 (c) 所示二叉树，按顺序读入字符 **a b # d e # # # c # #**，可建立相应的二叉链表。

程序 6-11 建立二叉链表算法

---

```

template<class T>
BinTreeNode<T> * BinaryTree<T>::CreateBinTree( )
{
    //按先序次序输入二叉树中结点的值（一个字符）
    // # 表示空树，构造用二叉链表表示的二叉树 T
    BinTreeNode<T> *p; T ch;
    cin>>ch;
    if (ch=='#') p=NULL;
    else
    {
        p=new BinTreeNode<T>;
        if (p==NULL) throw NoMem( );
        p->SetData(ch); p->SetLeftChild(CreateBinTree( ));
        p->SetRightChild(CreateBinTree( ));
    }
    return p;
}

```

---

在各种二叉树的遍历算法中，最基本的操作是访问结点。对一棵含  $n$  个结点的二叉树，任何一种遍历算法的时间复杂度均为  $O(n)$ 。遍历过程中所需要的辅助空间为栈的最大容量，即树的深度，在最坏情况下为  $n$ ，因此空间复杂度也为  $O(n)$ 。另外，还可采用其他存储结构实现二叉树的遍历，有些存储方法可通过省略栈来节约空间，但它们在时间上的损失又会很大，这里不再详细讨论。

6.6.2 二叉树的线索化

1. 定义

$n$  个结点的二叉链表中含有  $n+1$  个空指针域，利用二叉链表中的空指针域，存放指向结点在某种遍历次序下的前驱和后继结点的指针（这种附加的指针称为“线索”），则这种加上了线索的二叉链表称为线索链表，相应的二叉树称为线索二叉树（threaded binary tree）。线索链表解决了二叉链表能找到结点的左、右孩子，但无法直接找到该结点在某种遍历序列中的前驱和后继结点的问题。根据线索性质的不同，线索二叉树可分为前序线索二叉树、中序线索二叉树和后序线索二叉树三种。线索链表中的结点结构为：

lChild	lTag	data	rTag	rChild
--------	------	------	------	--------

其中：lTag = 0，lChild 域指示结点的左孩子；lTag = 1，lChild 域指示结点的前驱；  
rTag = 0，rChild 域指示结点的右孩子；rTag = 1，rChild 域指示结点的后继。

图 6.15（a）所示为中序线索二叉树，与其对应的中序线索链表如图 6.15（b）所示。其中实线为指针（指向左、右子树），虚线为线索（指向前驱和后继）。对二叉树以某种次序遍历使其变为线索二叉树的过程叫作线索化。

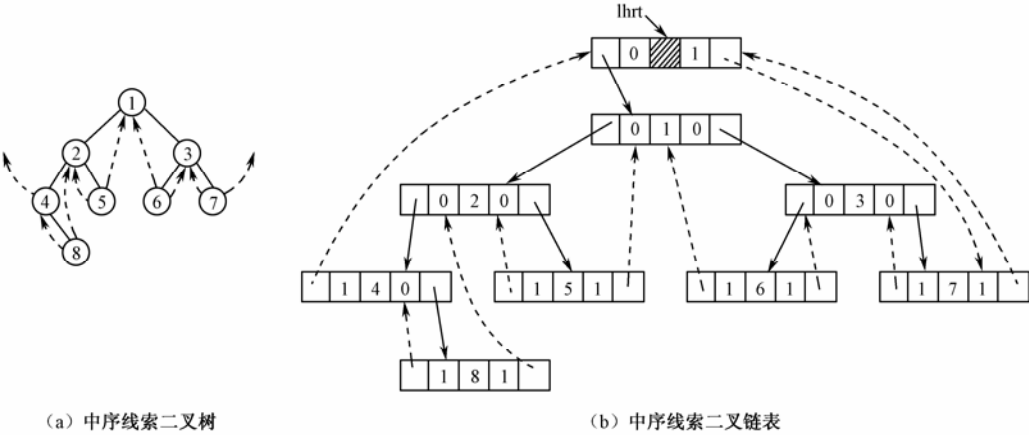


图 6.15 线索二叉树及其存储结构

2. 线索二叉树的结点类

通过中序线索二叉树进行二叉树的遍历比前面介绍的算法在时间和空间上更有效率。因此，当需要经常遍历二叉树或查找结点在遍历所得线性序列中的前驱和后继时，常采用线索链表作为存储结构。程序 6-12 和程序 6-13 给出了线索树二叉树结点类和树类的 C++ 语言描述。

程序 6-12 线索树二叉树结点的类描述

```
#include "TBinTreeNode.h"
enum PointerTag{Link, Thread}; //Link==0:指针, Thread==1:线索
template<class T>
class BinTreeThreadNode :public TBinTreeNode<T>
```

```

{
private:
    BinTreeThreadNode<T> *lc, *rc; //左、右孩子指针
    PointerTag LTag, RTag; //左、右标志
public:
    BinTreeThreadNode( ): lc(NULL), rc(NULL) {};
    virtual ~BinTreeThreadNode( ) {};
    BinTreeThreadNode(Td, BinTreeThreadNode<T> *lp=NULL,
    BinTreeThreadNode<T> *rp=NULL):data(d), lc(lp), rc(rp) {};
    //构造函数，构造一个数据域为 d 的结点
    virtual TBinTreeNode<T> *GetLeftChild( ) { return lc; };
    //读元素的左儿子指针
    virtual TBinTreeNode<T> *GetRightChild( ) { return rc; };
    //读元素的右儿子指针
    virtual void SetLeftChild(TBinTreeNode<T> *pChild)
    { lc=(BinTreeThreadNode<T> *)pChild; }; //置左儿子指针
    virtual void SetRightChild(TBinTreeNode<T> *pChild)
    { rc=(BinTreeThreadNode<T> *) pChild; }; //置右儿子指针
    virtual PointerTag GetLeftTag( ) { return LTag; };
    virtual PointerTag GetRightTag( ) { return RTag; };
    virtual bool SetLeftTag(PointerTag l);
    virtual bool SetRightTag(PointerTag r);
};

```

---

### 程序 6-13 线索二叉树的类定义

---

```

#include"BinaryTree.h"
#include"BinTreeThreadNode.h"
template<class T>
class BinaryThreadTree:public BinaryTree<T>
{
public:
    BinaryThreadTree( ) { tRoot=0; }; //构造函数
    BinaryThreadTree(TBinTreeNode<T> *p) //构造函数
    { root=(BinTreeThreadNode<T> *)p; };
    ~BinaryThreadTree( )//析构函数
    { if (tRoot) tRoot->SetRightChild(0); tRoot->SetLeftChild(0); };
    virtual TBinTreeNode<T> *CreateBinTree( ); //对 CreateBinTree 进行重载
    virtual void InOrderThreading( )
    { InOrderThreading((BinTreeThreadNode<T> *)root ); };
    virtual void InThreading(BinTreeThreadNode<T> *r);
    virtual InOrderTraverse_Thr( ) { InOrderTraverse_Thr(tRoot); };
private:
    BinTreeThreadNode<T> *tRoot, *pre;

```

```
virtual bool InOrderThreading(BinTreeThreadNode<T> *r);  
virtual bool InOrderTraverse_Thr(BinTreeThreadNode<T> *r );  
};
```

---

### 3. 遍历线索二叉树

对线索树的遍历，只要先找到序列中的第一个结点，然后依次找结点后继直至其后继为空时而结束。为方便起见，在二叉树的线索链表上也添加一个头结点，并令其 lChild 域的指针指向二叉树的根结点，其 rChild 域的指针指向中序遍历时访问的最后一个结点；同时令二叉树中序序列中的第一个结点的 lChild 域指针和最后一个结点 rChild 域的指针均指向头结点。这样就为二叉树建立了一个双向线索链表，既可从第一个结点起顺着后继进行遍历，也可从最后一个结点起顺着前驱进行遍历。程序 6-14 是以双向线索链表为存储结构时对二叉树进行遍历的算法。

程序 6-14 遍历线索二叉树算法

---

```
template<class T>  
bool BinaryThreadTree<T>::InOrderTraverse_Thr(BinTreeThreadNode<T> *r)  
{  
    if (r==NULL) return false;  
    BinTreeThreadNode<T> *p;  
    p=(BinTreeThreadNode<T> *)r->GetLeftChild( );  
    while (p!=r)  
    {  
        while (p->GetLeftTag( )==Link)  
            p=(BinTreeThreadNode<T> *)p->GetLeftChild( );  
        if (p==NULL) return false; cout<<p->GetData( )<<endl;  
        while (p->GetRightTag( )==Thread && p->GetRightChild( )!= r)  
        {  
            p=(BinTreeThreadNode<T> *) p->GetRightChild( );  
            cout<<p->GetData( )<<endl; //访问后继结点  
        }  
        p=(BinTreeThreadNode<T> *) p->GetRightChild( );  
    }  
    return true;  
}
```

---

### 4. 线索化二叉树

将二叉树线索化，实质上是将二叉树中的空指针改为指向其前驱或后继的线索，亦即在按某次序遍历二叉树的过程中修改指针，用线索取代空指针。为了记下遍历过程中访问结点的先后关系，附设一个指针 pre 始终指向刚刚访问过的结点。若指针 p 指向当前访问的结点，则 pre 指向它的前驱。由此可得中序遍历建立中序线索化链表的算法见程序 6-15。

程序 6-15 遍历线索二叉树算法

---

```
template<class T>  
bool BinaryThreadTree<T>::InOrderThreading(BinTreeThreadNode<T> *r)  
{  
    tRoot=new BinTreeThreadNode<T>;
```

```

    if (tRoot==NULL) return false;
    tRoot->SetLeftTag(Link);
    tRoot->SetRightTag(Thread); //建立头指针
    tRoot->SetRightChild(tRoot); //右指针回指
    if (r==NULL) tRoot->SetLeftChild(tRoot);
    //如果二叉树为空，则左指针回指
    else
    {
        tRoot->SetLeftChild(r); tRoot->SetLeftTag(Link);
        pre=tRoot; InThreading(r); //中序遍历进行中序线索化
        pre->SetRightChild(tRoot);
        pre->SetRightTag(Thread); //最后一个结点线索化
        tRoot->SetRightChild(pre);
    }
    return true;
}

template<class T>
void BinaryThreadTree<T>::InThreading(BinTreeThreadNode<T> *r)
{
    //利用中序遍历线索化以 r 为根的二叉树，pre 为中序序列中第一个结点的前驱
    if (r==NULL) return ;
    InThreading((BinTreeThreadNode<T> *)r->GetLeftChild( )); //左儿子线索化
    if (r->GetLeftChild( )==NULL)
    {
        r->SetLeftTag(Thread); r->SetLeftChild(pre); }
    if (pre->GetRightChild( )==NULL)
    {
        pre->SetRightTag(Thread); pre->SetRightChild(r); }
    pre=r;
    InThreading((BinTreeThreadNode<T> *)r->GetRightChild( ));
}

```

### 6.6.3 二叉树与森林的转换

树或森林与二叉树之间有自然的一一对应关系。任何一个森林或一棵树都可唯一地对应得到一棵二叉树；反之，任何一棵二叉树也能唯一地对应成一个森林或一棵树。从树的二叉链表表示可知，任何一棵树转换成二叉树后，其右子树必然为空，若将森林中第二棵树的根结点作为第一棵树根结点的兄弟便可使森林转换为二叉树，图 6.16 反映了森林与二叉树之间的对应关系。下面讨论它们之间的转换方法。

#### 1. 森林转换成二叉树

森林转换成二叉树的方法为：① 对于森林中的每一棵树，将其所有兄弟结点之间建立一条连线，并在所有的树根之间也建立一条连线；② 对于连接后的树中的每一个结点，除了保留该结点到它的最左子结点（即第一个子结点）的连线外，删除它到其他子结点的连线；③ 将经过上述步骤得到的树以根结点为轴，顺时针方向旋转  $45^\circ$ ，使其层次分明。

图 6.16 显示了森林转换为二叉树的过程。该方法的形式化描述如下。如果  $F = \{T_1, T_2, \dots, T_m\}$  是森林，则可按如下规则转换成一棵二叉树  $B = (\text{root}, \text{LB}, \text{RB})$ ：① 若  $F$  为空，



即  $m=0$ ，则  $B$  为空树；② 若  $F$  非空，即  $m \neq 0$ ，则  $B$  的根  $root$  即为森林中第一棵树的根  $ROOT(T_1)$ ； $B$  的左子树  $LB$  是从  $T_1$  中根结点的子树森林  $F_1 = \{T_{11}, T_{12}, \dots, T_{1m}\}$  转换而成的二叉树，其右子树  $RB$  是从森林  $F' = \{T_2, T_3, \dots, T_m\}$  转换而成的二叉树。

由森林与二叉树之间转换的关系可知，当森林转化成二叉树时，其第一棵树的子树森林转换成左子树，剩余树的森林转换成右子树，则上述森林的先序遍历和后序遍历即为其对应的二叉树的先序遍历和中序遍历。因此，当用二叉链表来存储时，树的先根遍历和后根遍历可借助于二叉树的先序遍历和中序遍历算法来实现。

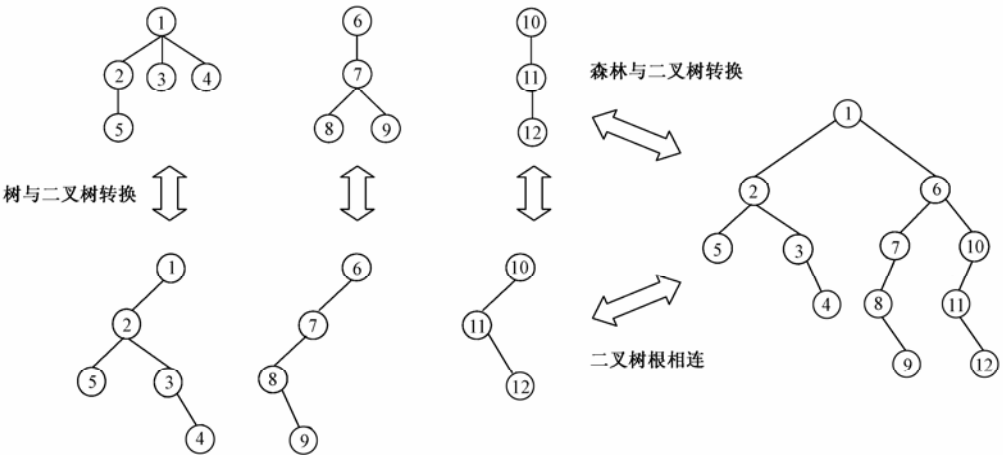


图 6.16 森林转换为二叉树的过程示意图

## 2. 二叉树转换成森林

二叉树转换成森林的方法为：① 若某结点是其双亲的左子结点，则把该结点的右子结点、右子结点的右子结点都与该结点的双亲结点用线连接起来；② 删去原二叉树中所有双亲结点与右子结点之间的连线；③ 对前面两个步骤所得到的结果进行旋转整理，使其结构层次分明。图 6.17 显示了二叉树转换为森林的过程。

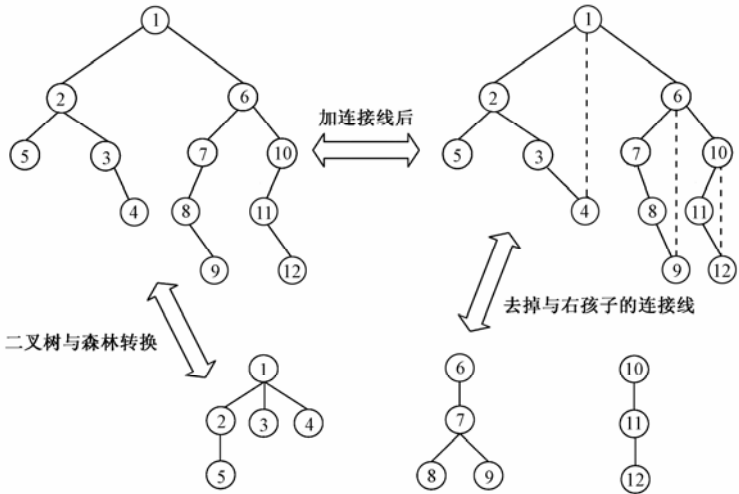


图 6.17 二叉树转换为森林的过程示意图

该方法的形式化描述如下。如果  $B=(\text{root}, \text{LB}, \text{RB})$  是一棵二叉树，则可按如下规则转换成森林  $F=\{T_1, T_2, \dots, T_m\}$ ：① 若  $B$  为空，则  $F$  为空；② 若  $B$  非空，则  $F$  中第一棵树  $T_1$  的根  $\text{ROOT}(T_1)$  即为二叉树  $B$  的根  $\text{root}$ ； $T_1$  中的根结点的子树森林  $F_1$  是由  $B$  的左子树  $\text{LB}$  转换而成的森林； $F$  中除  $T_1$  之外其余树组成的森林  $F'=\{T_2, T_3, \dots, T_m\}$  是由  $B$  的右子树  $\text{RB}$  转换而成的森林。从上述递归定义容易写出相互转换的递归算法。同时，森林和树的操作也可转换成二叉树的操作来实现。

## 6.7 哈夫曼树及其应用

哈夫曼（Huffman）树的定义：带权路径长度达到最小的二叉树即为哈夫曼树，也称为最优二叉树。

### 6.7.1 哈夫曼树

#### 1. 树的路径长度

树的路径长度是指从树根到树中每一结点的路径长度之和。在结点数目相同的二叉树中，完全二叉树的路径长度最短。

#### 2. 树的带权路径长度（weighted path length of tree，简记为WPL）

- ① 路径长度：从树中一个结点到另一个结点之间路径所包含的分支数目。
- ② 结点的权：在一些应用中赋予树中结点的一个具有某种意义的实数。
- ③ 结点的带权路径长度：从该结点到树根之间的路径长度与结点上权的乘积。
- ④ 树的带权路径长度（也称为树的代价）：树中所有叶结点的带权路径长度之和，通常记为

$$\text{WPL} = \sum_{i=1}^n w_i l_i$$

式中， $n$  表示叶子结点的数目， $w_i$  和  $l_i$  分别表示叶结点  $k_i$  的权值和根到结点  $k_i$  之间的路径长度。

#### 3. 最优二叉树或哈夫曼树

在由权值为  $w_1, w_2, \dots, w_n$  的  $n$  个叶子所构成的所有二叉树中，带权路径长度最小（即代价最小）的二叉树称为最优二叉树或哈夫曼树。

例如，在图 6.18 所示的 3 棵二叉树中，都有 4 个叶子结点  $x, y, z, w$ ，各结点有相应权值，它们的带权路径长度分别为：（a） $\text{WPL}=5\times 2+8\times 2+12\times 2+20\times 2=90$ ；（b） $\text{WPL}=20\times 1+12\times 2+5\times 3+8\times 3=83$ ；（c） $\text{WPL}=5\times 2+8\times 3+12\times 3+20\times 2=110$ 。

可见，尽管叶子结点及权相同，但相应的带权路径长度不同。其中（b）树的最小，可验证它就是哈夫曼树，即其带权路径长度在所有带权为 20、12、5、8 的 4 个叶子结点的二叉树中最小。注意：① 叶子上的权值均相同时，完全二叉树一定是最优二叉树；否则，完全二叉树不一定是在最优二叉树。② 最优二叉树中，权越大的叶子离根越近。③ 最优二叉树的形态不唯一，WPL 最小。

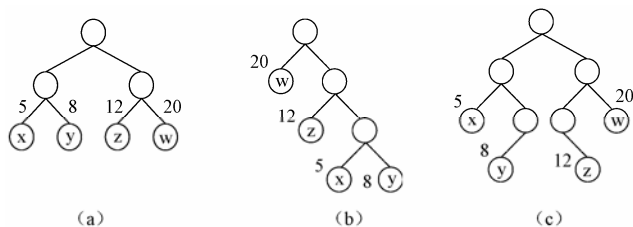


图 6.18 具有不同带权路径长度的二叉树

#### 4. 哈夫曼算法

哈夫曼最早给出了一个带有一般规律的算法，俗称哈夫曼算法，其描述如下：

- ① 根据给定的  $n$  个权值  $\{W_1, W_2, \dots, W_n\}$  构成  $n$  棵二叉树的集合  $F=\{T_1, T_2, \dots, T_n\}$ ，其中每棵二叉树  $T_i$  中只有一个权值为  $W_i$  的根结点，其左、右子树均为空。
- ② 在  $F$  中选取两棵根结点的权值最小的树作为左、右子树构造一棵新的二叉树，且新二叉树的根结点的权值为其左、右子树根结点的权值之和。
- ③ 在  $F$  中删除这两棵树，同时将新得到的二叉树加入到  $F$  中。
- ④ 重复②和③，直到  $F$  中只含一棵树为止，这棵树便是哈夫曼树。

例如，图 6.19 展示了图 6.18 (c) 所示的哈夫曼树的构造过程。其中，结点上标注的数字是所赋的权值。

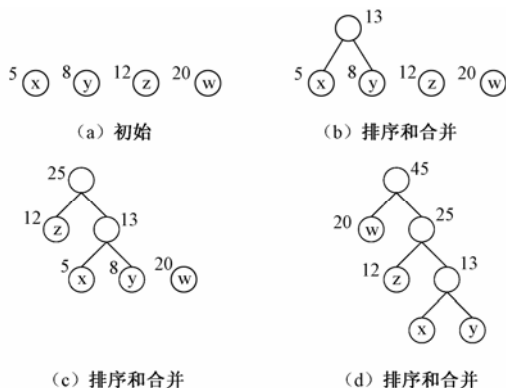


图 6.19 哈夫曼树的构造过程

#### 6.7.2 哈夫曼编码

哈夫曼编码的主要应用是实现数据压缩。例如，给出一段报文：CAST CAST SAT AT A TASA。字符集合是  $\{C, A, S, T\}$ ，各个字符出现的频度（次数）是  $W=\{2, 7, 4, 5\}$ 。若给每个字符以等长编码：A(00), T(10), C(01), S(11)，则总编码长度为  $(2+7+4+5) \times 2 = 36$ 。若按各个字符出现的概率不同而进行不等长编码，可望减小总编码长度。各字符出现概率为  $\{2/18, 7/18, 4/18, 5/18\}$ ，化整为  $\{2, 7, 4, 5\}$ 。以它们为各叶结点上的权值，建立哈夫曼树。左分支赋 0，右分支赋 1，得哈夫曼编码(变长编码)：A(0), T(10), C(110), S(111)，如图 6.20 所示。它的总编码长度为  $7 \times 1 + 5 \times 2 + (2+4) \times 3 = 35$ ，比等长编码的情形 (36) 要短，总编码

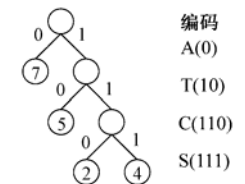


图 6.20 前缀编码示例

长度正好等于哈夫曼树的带权路径长度 WPL。哈夫曼编码是一种前缀编码。所谓前缀编码，是指任一个字符的编码都不是另一个字符的编码的前缀的编码。前缀编码解码时不会产生混淆。

下面讨论哈夫曼编码的具体实现方法。由哈夫曼树的特点可知，树中的所有结点的度要么为 2，要么为 0，即，不包含度为 1 的结点，则一棵带有  $n$  个叶子结点的哈夫曼树共有  $2n-1$  个结点。在已知叶子结点总数的情况下，可用一个大小为  $2n-1$  的一维数组来存储。对于结点结构，由于编码是通过在所构成的哈夫曼树上从叶子结点出发一直到根结点所经过的路径求得的，而译码则是从根结点出发一直到叶子所经过的路径来获得的，因此，对每个结点而言，采用既包含指向双亲结点的信息，又包含指向子结点的信息较为合适，由此给出下述存储结构（见程序 6-16～程序 6-20）。

程序 6-16 哈夫曼树存储的类描述

---

```
class HuffmanTree;
template<class T>
class HTNode{
    unsigned int  weight; //权值
    unsigned int  parent, lchild, rchild; //父亲、左儿子、右儿子指示器
public:
    friend HuffmanTree<T>;
};
```

---

程序 6-17 哈夫曼树类

---

```
template<class T>
class HuffmanTree
{
private:
    int *weight; //MaxSize 个字符的权值
    HTNode<T> *HTree; //哈夫曼树
    char **HuffmanCode; //MaxSize 个字符的哈夫曼编码
    int MaxSize; //最大字符个数
    int Select2Small(int i, unsigned int &s1, unsigned int &s2);
    //在 HT[1..i]中选择 parent 为 0 且 weight 最小的两个结点，其序号分别为 s1 和 s2
public:
    HuffmanTree(int size, int *w, unsigned int max); //构造函数
    ~HuffmanTree() { if (HTree) delete[] HTree; }; //析构函数
    void HuffmanCoding() { CreateHTree(); HCoding(); };
    void HuffmanCoding2() { CreateHTree(); HCoding2(); };
    void CreateHTree(); //建立哈夫曼树
    void HCoding(); //从叶子到根逆向求每个字符的哈夫曼编码
    void HCoding2(); //无栈非递归遍历哈夫曼树，求哈夫曼编码
    void print(); //输出哈夫曼树及编码
};
```

---

## 程序 6-18 构造函数

```
template<class T>
HuffmanTree<T>::HuffmanTree(int size=0, int *w=0)
{ MaxSize=size; weight=new int[MaxSize]; weight=w; }
```

## 程序 6-19 建立哈夫曼树

```
template<class T>
void HuffmanTree<T>::CreateHTree( )
{
    if (MaxSize<=0) return;
    int m=2*MaxSize-1;
    HTree=new HTNode<T>[m+1]; //0 号单元不用
    HTNode<T> *p;
    int i;
    for (p=HTree, i=1, p++; i<=MaxSize; ++i, ++p, ++weight)
    { (*p).weight=*weight; (*p).parent=0; (*p).lchild=0; (*p).rchild=0; }
    for (; i<=m; ++i, ++p)
    { (*p).weight=0; (*p).parent=0; (*p).lchild=0; (*p).rchild=0; }
    unsigned int s1, s2;
    for (i=MaxSize+1; i<=m; ++i)
    {
        //建哈夫曼树
        //在 HT[1..i-1]中选择 parent 为 0 且 weight 最小的两个结点，其序号分别为 s1 和 s2
        Select2Small(i-1, s1, s2);
        HTree[s1].parent=i; HTree[s2].parent=i; HTree[i].lchild=s1;
        HTree[i].rchild=s2; HTree[i].weight=HTree[s1].weight+HTree[s2].weight;
    }
}
```

## 程序 6-20 求哈夫曼编码

```
template<class T>
void HuffmanTree<T>::HCoding( )
{
    //从叶子到根逆向求每个字符的哈夫曼编码
    HuffmanCode=new char*[MaxSize+1]; //分配 n 个字符编码的头指针向量
    char *cd;
    cd=new char[MaxSize]; //分配求编码的工作空间
    cd[MaxSize-1]='\0'; //编码结束符
    int start; unsigned int c, f; int i;
    for (i=1; i<=MaxSize; ++i) { //逐个字符求哈夫曼编码
        start=MaxSize-1; //编码结束符位置
        for (c=i, f=HTree[i].parent; f!=0; c=f, f=HTree[f].parent)
            //从叶子到根逆向求编码
            if (HTree[f].lchild==c) cd[--start]='0'; else cd[--start]='1';
        HuffmanCode[i]=new char(MaxSize - start);
        //为第 i 个字符编码分配空间
    }
```

```
strcpy(HuffmanCode[i], &cd[start]); //从 cd 复制编码（串）到 HuffmanCode 中
} //for
delete [ ]cd; //释放工作空间
}
```

【例 6-4】 已知在一篇短文中仅包含 6 种不同的字符，各字符出现的次数分别为 4, 7, 12, 18, 25, 34。为节约存储空间，试为这 6 种字符设计哈夫曼编码。

(1) 构造哈夫曼树。根据 6 种字符出现的次数可建立哈夫曼树叶子结点的权重集合  $w = \{4, 7, 12, 18, 25, 34\}$ ， $n = 6$ 。按上述算法可构造一棵哈夫曼树如图 6.21 (a) 所示，各字符的哈夫曼编码如图 6.21 (b) 所示。

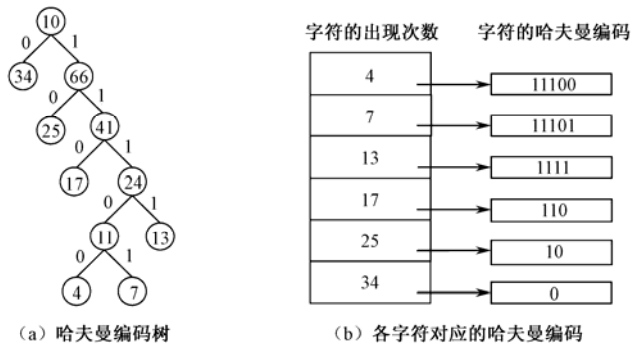


图 6.21 例 6-4 的图

(2) 哈夫曼树的带权路径长度值为  $(4+7) \times 5 + 13 \times 4 + 17 \times 3 + 25 \times 2 + 34 = 242$ ，即该文档原有的字符总数为  $4+7+12+18+25+34=100$ 。采用哈夫曼编码方式进行存储时，这 100 个字符所占用的空间为 242 位。而采用等长方式，每个字符需要 3 位，共需要 300 位。由此可见，哈夫曼编码能有效地节约空间。

【例 6-5】 试编写算法，对给定权值  $w$  构造哈夫曼树  $T$ 。

由于在哈夫曼树上没有度为 1 的结点，因此，有  $k$  个叶子的哈夫曼树有  $k-1$  个非终端结点，共有  $2k-1$  个结点。由于结点数已知且固定不变，可采用静态链表作为存储结构。设置一个大小为  $2k-1$  的数组，令数组的每个元素都由 4 个域组成，它们分别用于存储权值，双亲指针和左、右孩子指针，其类型定义如下：

Ltag	lchild	data	rchild	Rtag
------	--------	------	--------	------

```
struct node
{
    int wt; //权域
    int parent, lchild, rchild; //指针域
};
```

在这种存储结构上的哈夫曼算法如下：

```
const int maxint=...;
int m=...; int n;
void huffman(int k, int w[k], node t[2k-1]) //求给定权值 w 的哈夫曼树 t
{
    int i, j, x, y;
```

```

for (i=0; i<2k-1; i++) //置初态
{
    t[i].parent=-1; t[i].lchild=-1; t[i].rchild=-1;
    if (i<=m) t[i].wt=w[i]; else t[i].wt=0;
}
for (i=1; i<=k-1; i++) //构造新的二叉树
{
    x=0; y=0; m=maxint; n=maxint;
    for (j=0; j<k+i-1; j++) //找两棵权最小的二叉树
        if ((t[j].wt<m)&&(t[j].parent== -1))
            n=m; y=x; m=t[j].wt; x=j;
        else if ((t[j].wt<n)&&(t[j].parent== -1)) n=t[j].wt; y=j;
    t[x].parent=k+i; t[y].parent=k+i; //合并成一棵新的二叉树
    t[k+i].wt=m+n; t[k+i].lchild=x; t[k+i].rchild=y;
}
}

```

---

## 本章总结

### 1. 基本内容

本章主要介绍了二叉树的前序、中序、后序三种遍历算法和二叉树的线索化，树和森林的定义，树的存储结构及遍历，树、森林与二叉树之间的相互转换方法，哈夫曼树（最优二叉树）的构造方法及其应用。

### 2. 基本要求

(1) 掌握二叉树形结构的基本概念、术语、存储结构

- ① 清楚二叉树的逻辑结构、形式化定义、术语及其分层和非线性特性；
- ② 熟悉二叉树的基本形态、特点、基本运算；
- ③ 知道完全二叉树与满二叉树的定义、特点和区别；
- ④ 掌握二叉树的 5 条重要性质；
- ⑤ 了解二叉树的顺序存储结构的组织形式和实现方法；
- ⑥ 知道二叉链表的类型定义、结点形式及二叉链表的图示表示；
- ⑦ 了解三叉链表的结点形式和组织及实现方法。

(2) 掌握二叉树遍历及线索化的方法和算法

① 深入了解并掌握在二叉树上进行前序、中序和后序遍历的基本思想及其递归与非递归遍历算法；

- ② 深刻理解三种递归遍历算法（包括非递归算法）的执行过程及它们的异同；
- ③ 掌握二叉树线索化的基本思想和线索化方法及线索二叉树的表示形式（线索链表）。

(3) 知道树与森林的定义、存储方式及转换成二叉树的方法

① 清楚树的三种主要存储结构（双亲表示法、孩子表示法和孩子兄弟表示法）及其遍历方法；

- ② 了解树的计数、树与等价问题的定义和基本概念；

- ③ 熟练掌握树、森林和二叉树之间相互转换的方法。
- (4) 掌握树、二叉树、最优二叉树的综合应用程序设计方法
- 掌握构造哈夫曼树的方法及算法和哈夫曼编/译码的设计。

3. 重点与难点

本章重点是：树结构的术语、二叉树的定义、链式存储结构和遍历算法及线索化方法、树和森林与二叉树的转换关系。难点是：在二叉树上进行遍历的算法和综合应用程序设计（包括用递归方法与非递归方法实现的算法和程序）。

习题 6

6-1 分别就图 6.22 中的二叉树和树回答下列问题：① 哪个是根结点？② 哪些是叶子结点？③ 哪个是 G 的双亲？④ 哪些是 G 的祖先？⑤ 哪些是 E 的子孙？⑥ 哪些是 E 的兄弟？哪些是 C 的兄弟？⑦ 结点 B 和 I 的层数分别是多少？

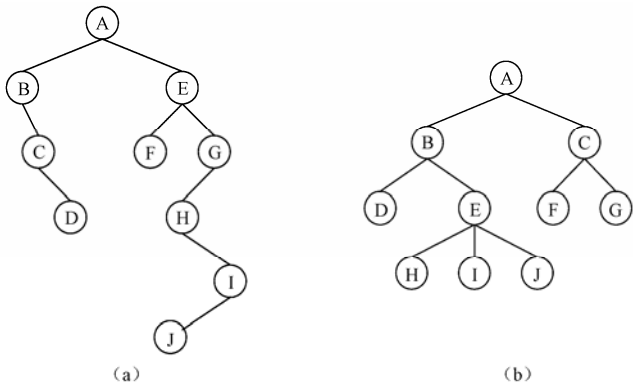


图 6.22 习题 6-1 的图

- 6-2 分别画出含有 3 个结点的无序树与二叉树的所有不同形态。
- 6-3 以二叉链表及三叉链表作为存储结构，分别实现二叉树的下列运算：① PARENT(BT, X)；② CREATE(X, LBT, RBT)；③ DELEFT(BT, X)。
- 6-4 分别写出图 6.22 (a) 所示二叉树的前序、中序和后序序列。
- 6-5 试找出分别满足下列条件的所有二叉树：① 前序序列和中序序列相同；② 中序序列和后序序列相同；③ 前序序列和后序序列相同。
- 6-6 以二叉链表作为存储结构，试编写求二叉树深度的算法。
- 6-7 一棵  $n$  个结点的完全二叉树存放在二叉树的顺序存储结构中，试编写非递归算法对该树进行前序遍历。
- 6-8 试编写算法判断两棵二叉树是否等价。满足下述条件，则称二叉树  $T_1$  和  $T_2$  是等价的： $T_1$  和  $T_2$  都是空的二叉树；或者  $T_1$  和  $T_2$  的根结点的值相同，并且  $T_1$  的左子树与  $T_2$  的左子树是等价的， $T_1$  的右子树与  $T_2$  的右子树是等价的。
- 6-9 试编写算法交换二叉树中所有结点的左、右子树（自选存储结构）。
- 6-10 试编写算法查找二叉链表中数据域值为  $x$  的结点（假定各结点的数据域值各不相同），并打印出  $x$  所有祖先的数据域值（提示：利用后序遍历非递归算法）。



6-11 已知一棵二叉树的中序序列和后序序列分别为 BDCEAFHG 和 DECBHGFA，试画出这棵二叉树。

6-12 试分别画出如图 6.23 所示树的孩子链表、孩子兄弟链表

和静态双亲链表。

6-13 试以孩子链表作为存储结构，实现树数据结构的下列运

算：① parent(T, x)；② child(T, x, i)；③ delete(T, x, i)。

6-14 试分别以孩子兄弟链表和静态双亲链表作为存储结构，

重做习题 6-13。

6-15 试分别给出图 6.23 所示树的前序、后序和层次遍历的结

点访问序列。

6-16 将如图 6.24 所示的森林转换成二叉树。

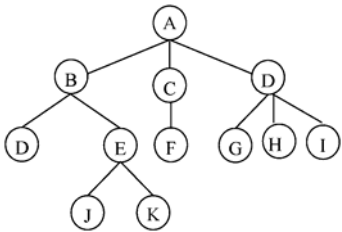


图 6.23 习题 6-12 的图

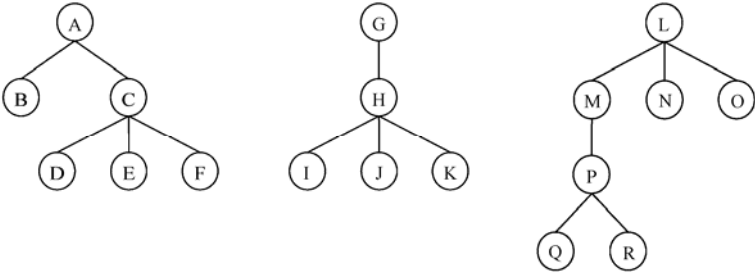


图 6.24 习题 6-16 的图

6-17 试分别编写二叉树中序遍历在下列存储结构上的非递归算法：

① 二叉链表；② 三叉链表（提示：考虑是否需要引入工作栈）。

6-18 试分别编写二叉树后序遍历在下列存储结构上的非递归算法：

① 二叉链表（提示：可在指针进栈的同时将一个标志进栈）；② 在存储结点中增设了标志的三叉链表（要求不用工作栈）。

6-19 画出如图 6.25 所示二叉树对应的森林。

6-20 给定权值 7, 18, 3, 32, 5, 26, 12, 8，构造相应的哈夫曼树。

6-21 试编写一个将百分制转换成五分制的算法，要求其时间性能尽可能高（即平均比较次数尽可能少）。假定学生成绩的分布情况如下：

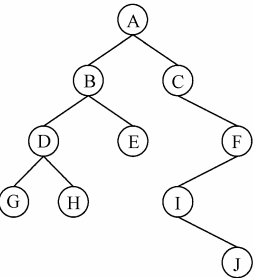


图 6.25 习题 6-19 的图

分数	0~59	60~69	70~79	80~89	90~100
比例	0.05	0.15	0.40	0.30	0.10

6-22 ① 若结点 A 有三个兄弟，而且 B 是 A 的父亲，试问结点 B 的度是多少？② 若某树有  $n_1$  个 1 度的结点， $n_2$  个 2 度的结点，……， $n_m$  个  $m$  度的结点，试问它有多少叶子结点。

6-23 在二叉树中查找值为  $x$  的结点，编写算法打印值为  $x$  的结点的所有祖先。假设值为  $x$  的结点不多于 1 个（提示：利用后序遍历非递归的算法，当找到  $x$  值的结点时，打印栈中有关的内容）。

6-24 中序和后序遍历一棵二叉树的结果分别为：中序 B, D, C, E, A, F, H, G 和后序 D, E, C, B, H, G, F, A。画出这样一个二叉树的逻辑结构和存储结构。

6-25 利用遍历的思想写一个将二叉树左、右孩子交换的算法（提示：不能用中序遍历）。

6-26 一棵  $n$  个结点的完全二叉树以向量作为其存储结构，试编写非递归算法，对该完全二叉树进行前序遍历。

6-27 假定  $T$  是一棵按标准形式存储的二叉树，试编写一个把  $T$  变成一棵线索树的程序过程。

6-28 假设二叉树  $T$  是一棵线索树，试编写一个按前序遍历树  $T$  的程序过程（不允许用递归过程，也不允许增加数组，只允许增加个别变量）。

6-29 试编写一个判断给定的二叉树是否为完全二叉树的程序过程。

6-30 如果  $T$  是结点值为整数的有序树， $T'$  是与  $T$  相对应的二叉树。假定  $T'$  已按标准形式存储。试编写以下程序过程：① 按前序打印树  $T$  的结点值；② 按后序打印树  $T$  的结点值；③ 打印树  $T'$  的叶子结点值；④ 求出树  $T'$  的度数。

6-31 试编写一个利用栈来实现后序遍历一棵给定二叉树的程序过程。

6-32 ① 将表达式  $((a+b)+c*(d+e)+f)*(g+h)$  改为：(a) 前缀表达式；(b) 后缀表达式。② 画出下列各表达式的树：(a)  $*a+b*c+de$ ；(b)  $*a+*b+cde$ 。

6-33 定义一个 ADT，它以二叉树为数学模型，具有操作：leftchild( $n$ )，rightchild( $n$ )，parent( $n$ ) 及 null( $n$ )。前三个操作分别求出结点  $n$  的左孩子、右孩子和父亲（若没有则返回  $\wedge$ ），最后一个操作的返回值为 true 当且仅当  $n$  为  $\wedge$ 。

6-34 试证：高度为  $h$  的二叉树中的结点个数至多为  $2^h-1$ 。具有  $2^h-1$  个结点，高为  $h$  的二叉树称为满二叉树。

6-35 设树  $T$  是一棵按标准形式存储的  $m$  次树，试编写一个获得树中所有叶子结点的程序过程。

6-36 ① 设树  $T$  是棵按标准形式存储的二叉树，试编写使用一个栈按前序遍历树中所有结点的程序过程。② 试编写使用一个栈按后序遍历树中所有结点的程序过程。

6-37 设树  $T$  是一棵有序树， $T_1$  是对应的二叉树，且  $T_1$  是用标准形式存储的，试编写以下程序过程：① 按前序遍历树  $T$  的所有结点；② 按后序遍历树  $T$  的所有结点；③ 遍历树  $T$  的所有叶子；④ 求出树  $T$  的次数。

6-38 由二叉树的前序和后序能否唯一地确定一棵二叉树？为什么？

## 第7章 图

图 (Graph) 是一种比树更复杂的非线性结构。在这种结构中, 任意两个结点之间都可能有关系, 即结点之间的连接关系是任意的, 因此图可用来描述更复杂的数据对象。图在人工智能、数学、物理、化学、电信、生物和计算机科学等领域都有着广泛的应用。本章利用图论的知识来讨论如何在计算机上实现图的操作, 主要学习图的存储结构以及图的操作实现等。

### 7.1 图的定义和术语

图的抽象数据类型包括它的数据结构和一组图的基本操作, 图的抽象数据类型定义见 ADT7-1。

ADT7-1 图的抽象类型定义

---

```
ADT Graph {  
    数据集合  $V$ :  $V$  是具有相同特性的数据元素的集合, 称为顶点集。  
    数据关系  $R$ :  $R=\{VR\}$ ;  $VR=\{<v,w>|v,w\in V \text{ 且 } P(v,w), <v,w>\text{表示从 } v \text{ 到 } w \text{ 的弧, 谓词 } P(v,w)\text{定义了弧 } <v,w>\text{的意义或信息}\}$   
    数据操作 P:  
    Create_Graph(&G, V, VR); //创建图  
        输入:  $V$  是图的顶点集,  $VR$  是图中弧的集合。  
        输出: 按照  $V$  和  $VR$  的定义构造图  $G$ 。  
    Destroy_Graph(&G); //删除图  
        输入: 图  $G$  存在。 输出: 删除图  $G$ 。  
    Locate_Vex(G, u); //判断顶点是否在图中  
        输入: 图  $G$  存在,  $u$  和  $G$  中顶点有相同特征。  
        输出: 若  $G$  中存在顶点  $u$ , 则返回该顶点在图中位置; 否则返回其他信息。  
    Get_Vex(G, v); //返回图中某顶点的值  
        输入: 图  $G$  存在,  $v$  是  $G$  中某个顶点。输出: 返回  $v$  的值。  
    Put_Vex(&G, v, value); //对图中某顶点赋值  
        输入: 图  $G$  存在,  $v$  是  $G$  中某个顶点。输出: 对  $v$  赋值  $value$ 。  
    First_AdjVex(G, v); //返回图中某顶点的第一个邻接顶点  
        输入: 图  $G$  存在,  $v$  是  $G$  中某个顶点。  
        输出: 返回  $v$  的第一个邻接顶点。若顶点在  $G$  中没有邻接顶点, 则返回“空”。  
    Insert_Vex(&G, v); //在图中增加顶点  
        输入: 图  $G$  存在,  $v$  和图中顶点有相同特征。输出: 在图  $G$  中增添新顶点  $v$ 。  
    Delete_Vex(&G, v); //在图中删除顶点  
        输入: 图  $G$  存在,  $v$  是  $G$  中某个顶点。  
        输出: 删除  $G$  中顶点  $v$  及其相关的弧。  
}
```

```

Insert_Arc(&G, v, w); //在图中增加边（弧）
    输入：图 G 存在，v 和 w 是 G 中两个顶点。
    输出：在图 G 中增添新弧<v, w>，若 G 是无向图，则还要增添对称弧<w, v>。
DeleteArc(&G, v, w); //在图中删除边（弧）
    输入：图 G 存在，v 和 w 是 G 中两个顶点。
    输出：在图 G 中删除弧<v, w>，若 G 是无向的，则还要删除对称弧<w, v>。
DFS_Traverse(G, visit()); //对进行深度优先遍历
    输入：图 G 存在，visit 是顶点的应用函数。
    输出：对图进行深度优先遍历。在遍历过程中，对每个顶点调用函数 visit() 一次
    且仅一次。一旦 visit() 失败，则操作失败。
BFS_Traverse(G, visit()); //对图进行广度优先遍历。
    输入：图 G 存在，visit 是顶点的应用函数。
    输出：对图进行广度优先遍历。在遍历过程中，对每个顶点调用函数 visit() 一次
    且仅一次。一旦 visit() 失败，则操作失败。
} ADT Graph

```

## （1）图的二元组定义

图  $G$  由两个集合  $V$  和  $E$  组成，记为  $G=(V, E)$ 。其中， $V$  是顶点（Vertex，图中的数据元素）的有穷非空集合， $E$  是  $V$  中顶点对（也称为边，表示两个顶点之间的关系）的有穷集。通常，也将图  $G$  的顶点集和边集分别记为  $V(G)$  和  $E(G)$ 。 $E(G)$  可以是空集，若  $E(G)$  为空，则图  $G$  只有顶点而没有边。

## （2）有向/无向图

① 有向图：若图  $G$  中的每条边都是有方向的，则称  $G$  为有向图（digraph）。有向边的表示：在有向图中，一条有向边是由两个顶点组成的有序对，有序对通常用尖括号表示。有向边也称为弧（arc），边的始点称为弧尾（tail），终点称为弧头（head）。例如， $\langle v_i, v_j \rangle$  表示一条有向边， $v_i$  是边的始点（起点）， $v_j$  是边的终点。因此， $\langle v_i, v_j \rangle$  和  $\langle v_j, v_i \rangle$  是两条不同的有向边。

② 无向图：若图  $G$  中的每条边都是没有方向的，则称  $G$  为无向图（undigraph）。无向图中的边均是顶点的无序对，无序对通常用圆括号表示。例如，无序对  $(v_i, v_j)$  和  $(v_j, v_i)$  表示同一条边。

例如，图 7.1（a）中， $G_1$  是有向图，定义此图的谓词  $P(v, w)$  则表示从  $v$  到  $w$  的一条单向通路。 $G_1=(V_1, \{A_1\})$ ，其中  $V_1=\{v_1, v_2, v_3, v_4\}$ ， $A_1=\{\langle v_1, v_2 \rangle, \langle v_1, v_3 \rangle, \langle v_2, v_1 \rangle, \langle v_2, v_3 \rangle, \langle v_2, v_4 \rangle, \langle v_3, v_4 \rangle, \langle v_4, v_1 \rangle\}$ ；图 7.1（b）中， $G_2$  为无向图  $G_2=(V_2, \{A_2\})$ ，其中  $V_2=\{v_1, v_2, v_3, v_4, v_5\}$ ， $A_2=\{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_1, v_5), \langle v_2, v_3 \rangle, (v_2, v_5), (v_3, v_4), (v_3, v_5), \langle v_4, v_5 \rangle\}$ 。

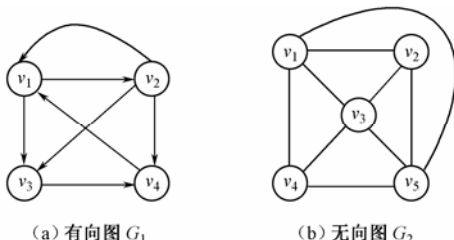


图 7.1 图的示例

### (3) 完全图

假设用  $n$  表示图中顶点数目, 用  $e$  表示边或弧的数目, 并不考虑顶点到其自身的弧或边, 即若  $\langle v_i, v_j \rangle \in E$ , 则  $v_i \neq v_j$ , 对于无向图, 其边数  $e$  的取值范围是  $0 \sim n(n-1)/2$ , 当  $e=n(n-1)/2$  时, 该无向图称为完全图 (completed graph); 对于有向图, 其边数  $e$  的取值范围是  $0 \sim n(n-1)$ , 当  $e=n(n-1)$  时, 该有向图称为有向完全图; 边数或弧 (如  $e < n \lg n$ ) 非常少的图称为稀疏图 (sparse graph), 反之称为稠密图 (dense graph)。

### (4) 权

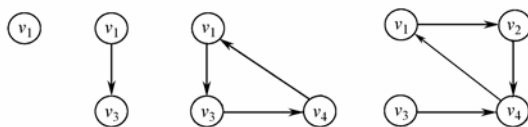
与图的边或弧相关的数称为权 (weight), 权值通常表示从一个顶点到另一个顶点的距离或耗费, 带权的图通常称为网 (network)。

### (5) 子图

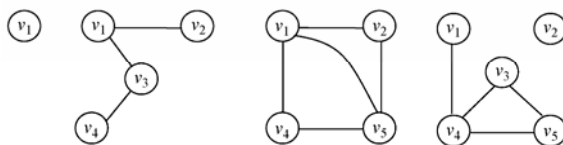
假设两个图  $G=(V, \{E\})$  和  $G'=(V', \{E'\})$ , 如果  $V'$  是  $V$  的子集, 且  $E'$  是  $E$  的子集, 则称  $G'$  为  $G$  的子图 (subgraph)。例如, 子图的一些例子如图 7.2 所示。

### (6) 度

对于无向图  $G=(V, \{E\})$ , 如果边  $(v, v') \in E$ , 则称顶点  $v$  和  $v'$  互为邻接点 (adjacent), 即  $v$  和  $v'$  相邻接。边  $(v, v')$  依附 (incident) 于顶点  $v$  和  $v'$ , 或者说  $(v, v')$  和顶点  $v$  和  $v'$  相关联。顶点  $v$  的度 (degree) 是和  $v$  相关联的边的数目, 记为  $TD(v)$ 。例如,  $G_2$  中顶点  $v_3$  的度是 4。



(a) 图 7.1 中  $G_1$  的子图



(b) 图 7.1 中  $G_2$  的子图

图 7.2 子图示例

对于有向图  $G=(V, \{A\})$ , 如果弧  $\langle v, v' \rangle \in A$ , 则称顶点  $v$  邻接到顶点  $v'$ , 顶点  $v'$  邻接自顶点  $v$ 。弧  $\langle v, v' \rangle$  和顶点  $v, v'$  相关联。以顶点  $v$  为头的弧的数目称为  $v$  的入度 (in degree), 记为  $ID(v)$ ; 以  $v$  为尾的弧的数目称为  $v$  的出度 (out degree), 记为  $OD(v)$ ; 顶点  $v$  的度为  $TD(v) = ID(v) + OD(v)$ 。例如, 图  $G_1$  中顶点  $v_2$  的入度  $ID(v_2)=1$ , 出度  $OD(v_2)=3$ , 度  $TD(v_2)=ID(v_2)+OD(v_2)=4$ 。一般地, 如果顶点  $v_i$  的度记为  $TD(v_i)$ , 那么一个有  $n$  个顶点,  $e$  条边或弧的图, 满足如下关系

$$e = \frac{1}{2} \sum_{i=1}^n TD(v_i)$$

### (7) 路径

无向图  $G=(V, \{E\})$  中从顶点  $v$  到顶点  $v'$  的路径 (path) 是一个顶点序列  $(v=v_0, v_1, \dots, v_m=v')$ , 其中  $(v_{j-1}, v_j) \in E, 1 \leq j \leq m$ 。如果  $G$  是有向图, 则路径也是有向的, 顶点序列应满足  $\langle v_{j-1}, v_j \rangle \in E, 1 \leq j \leq m$ , 路径的长度是路径上的边或弧的数目。当第一个顶点和

最后一个顶点相同时，路径称为回路或环（cycle）；当路径中的顶点序列不出现重复值时，该路径称为简单路径；除了第一个顶点和最后一个顶点之外的其余顶点不重复出现的回路，称为简单回路或简单环。

### （8）连通

在无向图  $G$  中，如果从顶点  $v$  到顶点  $v'$  有路径，则称  $v$  和  $v'$  是连通的。如果对于图中任意两个顶点  $v_i, v_j \in V$ ,  $v_i$  和  $v_j$  都是连通的，则称  $G$  是连通图（connected graph）。图 7.1（b）中的  $G_2$  就是一个连通图，而图 7.3（a）中的  $G_3$  则是非连通图，但  $G_3$  有两个连通分量，如图 7.3（b）所示。所谓连通分量（connected component），是指无向图中的极大连通子图。

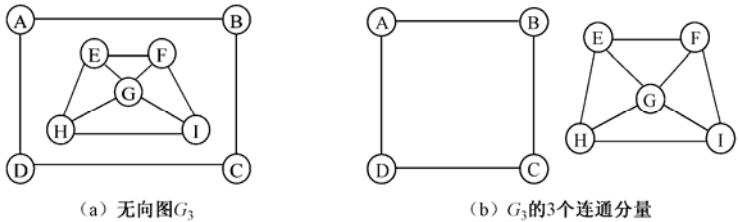


图 7.3 无向图及其连通分量

在有向图  $G$  中，如果对于每一对  $v_i, v_j \in V, v_i \neq v_j$ , 从  $v_i$  到  $v_j$  和从  $v_j$  到  $v_i$  都存在路径，则称  $G$  是强连通图。有向图中的极大强连通子图称作有向图的强连通分量。例如，图 7.1（a）中的  $G_1$  不是强连通图，但它有两个强连通分量，如图 7.4 所示。

一个连通图的生成树是一个极小连通子图，它含有图中全部顶点，但只有足以构成一棵树的  $n-1$  条边。如果在一棵生成树上添加一条边，则构成一个环。一棵有  $n$  个顶点的生成树有且仅有  $n-1$  条边。如果一个图有  $n$  个顶点和小于  $n-1$  条边，则是非连通图。如果它多于  $n-1$  条边，则一定有环。但是，有  $n-1$  条边的图不一定是生成树。图 7.5 是图 7.1（b）中  $G_2$  的一棵生成树。

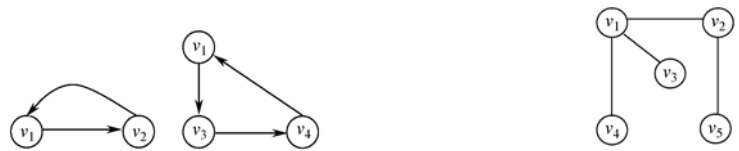


图 7.5  $G_2$  的一棵生成树

如果一个有向图恰有一个顶点的入度为 0，其余顶点的入度均为 1，则是一棵有向树。一个有向图的生成森林由若干棵有向树组成，含有图中全部顶点，但只有足以构成若干棵不相交的有向树的弧。如图 7.6 所示。

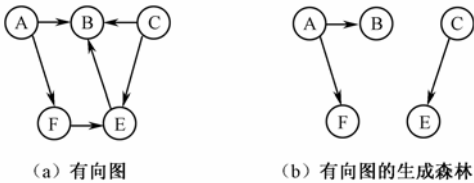


图 7.6 一个有向图及其生成森林

## 7.2 图的对象抽象模型

由于图结点之间的关系比较复杂，因此它的抽象结构也较为复杂。这里首先定义一个常量，以方便后面使用：

---

```
//定义常量，表示最大结点个数，实际结点个数可小于该数
const MAX_VERTEX_NUM=100;
```

---

### 7.2.1 图结点对象抽象模型

图结点对象抽象模型的重点是描述图结点的内容，隐蔽其具体结构，因此对象模型的重点主要是关于结点信息的 **Get** 和 **Set** 类操作。程序 7-1 为图结点的类描述，其中成员函数均已实现。

程序 7-1 图结点的类描述

---

```
template <class T> //结点内容的类型（可变类型）
class GraphNode //图结点类
{
protected:
    long no;    //图结点的编号
    float weight; //结点的权
    T info;    //图结点的内容
public:
    GraphNode( ) { no=0; weight=0; };
    virtual long GetNo( ) { return no; }; //返回本结点的编号
    virtual void SetNo(long mNo) { no=mNo; }; //设置编号值为 mNo
    virtual T& GetInfo( ) { return info; }; //返回本结点的值
    virtual T& SetInfo(T &e) { info=e; return info; }; //设置结点内容为 e
    virtual float GetWeight( ) { return weight; }; //返回本结点的权值
    virtual void SetWeight(float w) { weight=w; }; //设置本结点的权值为 w
};
```

---

### 7.2.2 图的边对象抽象模型

图的边不仅体现图中数据元素之间的关系，而且其本身也具有诸如权之类的属性，因此需要建立其对象模型。图的边的类描述见程序 7-2。

程序 7-2 图的边的类描述

---

```
#define INFINITY 10000 //表示两个定点不连通
template <class T>
class GraphEdge //图边抽象类
{
protected:
    int startNo, endNo; //边的起点和终点
```

```

float weight; //边权
T info; //边的内容
public:
    GraphEdge( ) { startNo=-1; endNo=-1; weight=INFINITY; info=0; };
    virtual T &GetInfo( ) { return info; }; //读边内容
    virtual void SetInfo(T &e) { info=e; }; //设置边内容为 e
    virtual float GetWeight( ) { return weight; }; //读边的权
    virtual void SetWeight(float w) { weight=w; }; //设置边权
    virtual long GetStartNo( ) { return startNo; }; //读边的起点编号
    virtual long GetEndNo( ) { return endNo; }; //读边的终点编号
    virtual void SetStartNo(long mNo) { startNo=mNo; }; //设置边的起点编号
    virtual void SetEndNo(long mNo) { endNo=mNo; }; //设置边的终点编号
};

```

---

### 7.2.3 图对象抽象模型

图的对象抽象模型的重点在于定义图的整体结构操作，这些操作主要包括遍历、查找、求路径、求生成树、求拓扑序列（有向图）、求关键路径、求最短路径、求最小生成树等，程序 7-3 为具体的抽象模型描述。

程序 7-3 图的类描述

---

```

template <class VertexType, class EdgeType>
//用到两个可变类型，分别是图结点内容和边内容
class Graph //图抽象类
{
protected:
    long numNodes; //当前的顶点数
    long numEdges; //当前的边数
    long maxNodes; //最大顶点数
public:
    virtual bool FindVertexInfo(const VertexType vertexInfo)=0;
    //查找图中顶点 v 是否存在
    virtual bool LocateVertexInfo(const VertexType vertexInfo, long &pos)=0;
    //取顶点 v 在数组中的位置 pos
    virtual bool FindVertexNo(const long vertexNo)=0;
    //查找图中编号为 vertexNo 的顶点是否存在
    virtual bool LocateVertexNo(const long vertexNo, long &pos)=0;
    //取编号为 vertexNo 的顶点在数组中的位置 pos
    int IsEmpty( ) const{ return numNodes; }; //判断图是否为空
    long NumOfVertexes( ) { return numNodes; }; //取图的顶点数
    long NumOfEdges( ) { return numEdges; }; //取图的边数
    virtual bool GetEdgeWeight(long v1, long v2, float &w)=0;
    //取边<v1, v2>上的权;
    virtual long GetFirstNeighbor(long v)=0;

```



```

//取图中顶点 v 的第一个邻接点的序号，不存在则返回-1
virtual long GetNextNeighbor(long v1, long v2)=0;
//取图中顶点 v1 在 v2 之后的下一个邻接点的序号，不存在则返回-1
virtual bool GetVertex(long i, GraphNode<VertexType> &node);
//获得第 i 个结点，成功则返回 true
virtual int SetVertex(long i, GraphNode<VertexType> *node);
//出错则返回-1，如果 i 不存在，则新增并返回 0，设置成功则返回 1
virtual bool DeleteVertex(long v)=0; //删除编号为 v 及依附于 v 的边
virtual bool DeleteEdge(long v1, long v2)=0;
//在图中删除依附于编号为 v2 和 v2 的顶点的边
virtual long GetPrior(long vertexNo, int n)=0;
//取编号为 vertexNo 结点的编号最小的前驱，不存在则返回-1，出错则返回-2
virtual int GetOutDegree(long vertexNo)=0;
//返回编号为 vertexNo 结点的出度
virtual int GetInDegree(long vertexNo)=0;
//返回编号为 vertexNo 结点的入度
virtual bool SetEdge(long v1, long v2, EdgeType &info, float w)=0;
//...
}

```

## 7.3 图的存储结构

图的存储表示方法最常用的有 4 种，分别是邻接矩阵、邻接表、邻接多重表和十字链表表示法，在使用过程中要根据具体的应用特点和相关操作来进行选择。

### 7.3.1 邻接矩阵

#### 1. 存储方法描述

邻接矩阵（adjacency matrix）是一种用边的集合表示图的方法，其中边的集合用一个二维数组表示，数组的每个元素表示一条边，元素的两个下标值分别表示边的两个端点的编号，结点编号与数组的下标相对应。图的邻接矩阵可定义为

$$A[i, j] = \begin{cases} 1 & (v_i, v_j) \text{ 或 } <v_i, v_j> \in VR \\ 0 & \text{反之} \end{cases}$$

例如，图 7.1 中， $G_1$  和  $G_2$  的邻接矩阵如图 7.7 所示。以二维数组表示具有  $n$  个顶点的图时，需存放  $n$  个顶点信息和  $n^2$  个弧信息的存储量。若考虑无向图邻接矩阵的对称性，则可采用压缩存储方式只存入矩阵的下三角（或上三角）元素。

通过邻接矩阵，很容易判断任意两个顶点之间是否有边（或弧）相连，同时，也可以很方便地求得各个顶点的度。对于无向图，顶点  $i$  的度是邻接矩阵中第  $i$  行（或第  $i$  列）的元素之和，即  $TD(v_i) = \sum_{j=0}^{n-1} A[i][j] (n = \text{MAX\_VERTEX\_NUM})$ 。

$$G_1.\text{arcs} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad G_2.\text{arcs} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

图 7.7 图 7.1 中  $G_1$ 、 $G_2$  的邻接矩阵

对于有向图，第  $i$  行的元素之和为顶点  $v_i$  的出度  $\text{OD}(v_i)$ ，第  $j$  列的元素之和为顶点  $v_j$  的入度  $\text{ID}(v_j)$ 。网的邻接矩阵可定义为

$$A[i, j] = \begin{cases} w_{i,j} & (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \in VR \\ \infty & \text{反之} \end{cases}$$

如图 7.8 所示为一个有向网和它的邻接矩阵。

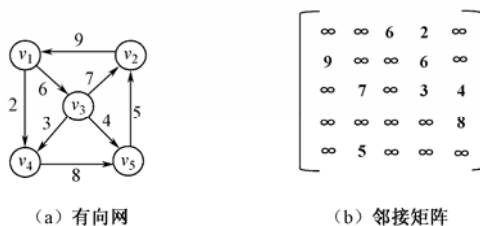


图 7.8 有向网及其邻接矩阵

## 2. 相关类的描述

若用二维数组  $A$  表示图，则  $A[i][j]$  表示边  $\langle i, j \rangle$  或  $(i, j)$ ，由于前面给出的边的抽象定义 **GraphEdge** 和结点抽象定义 **GraphNode** 中包含了邻接矩阵表示法中所需要的相关信息，因此邻接矩阵的边和结点可由 **GraphEdge** 类和 **GraphNode** 类直接派生（未增加新成员）。程序 7-4～程序 7-6 分别给出了邻接矩阵的相关类描述。

程序 7-4 邻接矩阵边的类描述

```
template <class TEdgeInfo>
class GraphEdgeMatrix:public GraphEdge <TEdgeInfo>
{ public:
//邻接矩阵边，GraphEdge 中的函数均已实现，故这里不重载时不需给出
};
```

程序 7-5 邻接矩阵结点的类描述

```
template <class T>
class GraphNodeMatrix:public GraphNode<T> //邻接矩阵结点抽象类
{ //GraphNode 中的函数均已实现，故这里不重载时不需给出
//下面可声明 GraphNodeMatrix 中特有的函数
};
```

程序 7-6 邻接矩阵图的类描述

```
template <class VertexType, class EdgeType>
//用到两个可变类型，分别是图结点内容和边内容
class GraphMatrix:public Graph<EdgeType, VertexType>
```

```

{ //邻接矩阵类
protected:
    long maxNodes; //最大顶点数
    GraphEdgeMatrix<EdgeType>
    edges[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
    //存放图边的矩阵
    GraphNodeMatrix<VertexType> nodes[MAX_VERTEX_NUM];
    //存放图结点的一维数组
    int smpEdges[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
    //记录有向图信息的矩阵
public:
    GraphMatrix(int num=MAX_VERTEX_NUM);
    virtual bool FindVetexInfo(const VertexType vertexInfo);
    //查找图中顶点 v 是否存在
    virtual bool LocatVertexInfo(const VertexType vertexInfo, long &pos);
    //取顶点 v 在数组中的位置 pos
    virtual bool FindVertexNo(const long vertexNo);
    //查找图中编号为 vertexNo 的顶点是否存在
    virtual bool LocateVertexNo(const long vertexNo, long &pos);
    //取编号为 vertexNo 的顶点在数组中的位置 pos
    virtual bool GetEdgeWeight(long v1, long v2, float &w);
    //取边<v1, v2>上的权;
    virtual long GetFirstNeighbor(long v);
    //取图中顶点 v 的第一个邻接点的序号, 不存在返回-1
    virtual long GetNextNeighbor(long v1, long v2);
    //取图中顶点 v1 在 v2 之后的下一个邻接点的序号, 不存在返回-1
    virtual bool DeleteVertex(long v); //删除编号为 v 及依附于 v 的边
    virtual bool DeleteEdge(long v1, long v2);
    //在图中删除依附于编号为 v1 和 v2 的顶点的边
    virtual bool GetVertex(long i, GraphNode<VertexType> &node);
    //获得第 i 个结点
    virtual int SetVertex(long i, GraphNode<VertexType> *node);
    //设置第 i 个结点, 如果不存在, 则插入到最后
    virtual long GetPrior(long vertexNo, int n);
    //取编号为 vertexNo 结点的编号最小的前驱, 如果没有则返回-1, 出错则返回-2
    virtual bool SetEdge(long v1, long v2, EdgeType &info, float w);
    virtual bool SetAEdge(long v1, long v2, EdgeType &info, float w) { return 0; };
    //下面为遍历函数, 在后面介绍
    virtual long DFS1(int g[][MAX_VERTEX_NUM], long n, long v0, char
        *visited, long *resu, long &top);
    virtual long DFS1(long v0, long *resu);
    virtual void DFSTraverse(long *resu);
    virtual long DFS1_NR(int g[][MAX_VERTEX_NUM], long n, long v0, char

```

```

        *visited, long *resu);
virtual long DFS1_NR(long v0, long *resu);
virtual long BFS1_NR(int g[ ][MAX_VERTEX_NUM], long n, long v0, char
        *visited, long *nos);
virtual long BFS1_NR(long v0, long *nos);
virtual void BFS1_NRTraverse(long *resu);
//...
};

```

---

如果图中结点的个数变化很大，则边矩阵可考虑用动态存储区（动态二维数组）来实现。由于标准的 C/C++ 语言不直接支持该功能，所以需要 `new()` 函数申请一块连续空间，然后将其封装在一个类中，并设置二维数组访问接口。

### 3. 算法实现

下面考虑在邻接矩阵存储方式下，图的基本操作的实现方法。

#### (1) 结点、边的增删

对邻接矩阵，结点或边的增删的算法较为简单，但一般要移动元素。

- 删除边：删除一条边比较简单，对相应矩阵元素设置无边标志即可。
- 删除点：删除一个结点，要将与它关联的边一同删除，即删除结点在矩阵中对应的行与列。
- 增加边：为现有两个结点间增加一条边，对相应矩阵元素设置边信息即可。
- 增加点：增加一个新结点，要在矩阵中相应位置插入一行一列。
- 增加新结点边：即插入带新结点的边，需先插入新结点，然后设置边信息。

#### (2) 度的计算

- 出度：结点  $i$  的出度等于  $i$  所对应的行上的非特殊元素的个数。
- 入度：结点  $i$  的入度等于  $i$  所对应的列上的非特殊元素的个数。

#### (3) 找邻接点/判别是否有边

判别邻接矩阵中结点  $i$  到结点  $j$  之间是否有边，只需查看元素  $A[i][j]$  的值即可；而查找某个结点的邻接点，则需要依次判别该结点与其他结点之间是否有边。

#### (4) 找路径

$\text{Path}(x, y)$  表示找出结点  $x$  到结点  $y$  之间的路径，程序 7-7 的伪码描述了它的算法。

程序 7-7 找路径算法

---

```

Path(x, y)
{
    if (x==y) return x; //所找的路径为(x)
    if (若 x 到 y 有边) return(x, y); //所找的路径为(x, y)
    for (x 的每个直接可达点 u)
    {
        if (u 尚未试探过)
            if (Path(u, y) 存在) return(x, u)+Path(u, y);
    }
    return 0; //空路径
}

```

---

该算法并未涉及图的具体存储结构，因此适用于任意存储结构。对不同的存储结构，只是在判别是否有边以及找邻接点的方法上有所不同。程序 7-8 给出邻接矩阵类的部分成员函数的实现。

程序 7-8 邻接矩阵图的类的部分成员函数

```
template <class VertexType, class EdgeType>
GraphMatrix<VertexType, EdgeType>::GraphMatrix(int num=MAX_VERTEX_NUM)
{   if (num>MAX_VERTEX_NUM)   throw OutOfBounds( );
    //如果数目超出范围，则抛出异常
    numNodes=0; //顶点数目
    maxNodes=num; for (int i=0; i<num; i++) //邻接矩阵初始化
        for (int j=0; j<num; j++) {   smpEdges[i][j]=0; }
    numEdges=0; //当前边数为 0
}

template <class VertexType, class EdgeType>
bool GraphMatrix<VertexType, EdgeType>::SetEdge(long v1, long v2, EdgeType
&info, float w)
{   if (v1<0 || v2<0)   return false;
    if (!FindVertexNo(v1)|| !FindVertexNo(v2))   return false;
    long pos1, pos2;
    if (LocateVertexNo(v1, pos1)&& LocateVertexNo(v2, pos2))
    {   if (edges[pos1][pos2].GetStartNo( )== -1 ||
        edges[pos1][pos2].GetEndNo( )== -1)   {   numEdges++; }
        edges[pos1][pos2].SetStartNo(v1); edges[pos1][pos2].SetEndNo(v2);
        edges[pos1][pos2].SetWeight(w); edges[pos1][pos2].SetInfo(info);
        if (w==0) smpEdges[pos1][pos2]=1; else   smpEdges[pos1][pos2]=w;
    }
    return true;
}

template <class VertexType, class EdgeType>
int GraphMatrix<VertexType, EdgeType>::SetVertex(long i,
GraphNode<VertexType> *node)
{   if (i<0)   return -1;
    if (i>=numNodes && (numNodes+1)<= maxNodes)
    {   nodes[numNodes]=*((GraphNodeMatrix<VertexType>*)node);
        numNodes++;   return 0;
    }
    else {   nodes[i]=*((GraphNodeMatrix<VertexType>*)node); return 1; }
}

template <class VertexType, class EdgeType>
```

```
bool GraphMatrix<VertexType, EdgeType>::FindVertexNo(const long vertexNo)
{
    for (long i=0; i<numNodes; i++)
    {
        if (nodes[i].GetNo( )==vertexNo) return true;
    }
    return false;
}

template <class VertexType, class EdgeType>
bool GraphMatrix<VertexType, EdgeType>::GetEdgeWeight(long v1, long
v2, float &w)
{
    if (v1<0 || v2<0) return false;
    if (!FindVertexNo(v1)|| !FindVertexNo(v2)) return false;
    long pos1, pos2;
    if (LocateVertexNo(v1, pos1)&& LocateVertexNo(v2, pos2))
    {
        w=edges[pos1][pos2].GetWeight( ); return true;
    }
    w=0; return false;
}
...

```

7.3.2 邻接表

1. 存储方法描述

邻接表（adjacency list）是图的顺序存储和链式存储相结合的一种存储方法，它对图  $G$  中的每个顶点  $v_i$  建立一个单链表，将所有邻接于  $v_i$  的顶点放到该链表中。每个结点包含三个域：邻接点域（adjvex），指示与顶点  $v_i$  邻接的点的连接关系；链域（nextarc），指示下一条边或弧的结点；数据域（info），存储与边或弧相关的信息，如权值等。每个链表上附设一个头结点，该结点除了设有链域（firstarc）指向链表中第一个结点之外，还设有存储顶点  $v_i$  的名称或其他有关信息的数据域（data）。头结点通常采用顺序结构进行存储，以便进行随机访问。图 7.9 描述了两中结点的结构。



例如，图 7.9（a）和（b）所示分别为图 7.1 中  $G_1$  和  $G_2$  的邻接表。

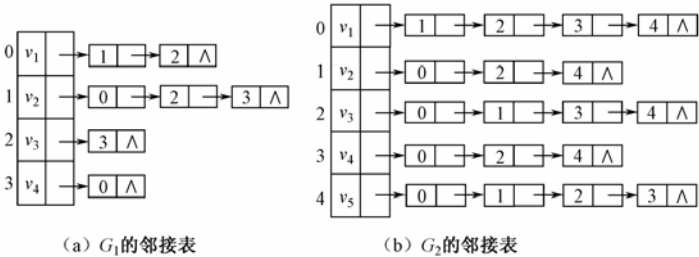


图 7.9 邻接表

在无向图的邻接表中，顶点  $v_i$  的度即为第  $i$  个链表中的结点数；而在有向图中，第  $i$  个链表中的结点数只是该顶点的出度，为求入度，还必须遍历整个邻接表，在所有链表中，

其邻接点域的值为  $i$  的结点的个数是顶点  $v_i$  的入度。为了便于确定顶点的入度或以顶点  $v_i$  为头的弧，可建立一个有向图的逆邻接表，即对每个顶点  $v_i$  建立一个链接以  $v_i$  为头的弧的表。例如，图 7.10 所示为图 7.1 中有向图  $G_1$  的逆邻接表。在建立邻接表或逆邻接表时，如果顶点的编号和顶点的数据值相同，则建立邻接表的复杂度为  $O(n+e)$ ，否则，需要通过查找才能得到顶点在图中的位置。

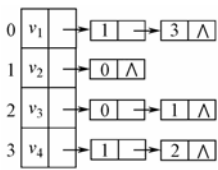


图 7.10  $G_1$  的逆邻接表

在邻接表上找任意一个顶点的第一个邻接点和下一个邻接点比较容易，但要判定任意两个顶点  $v_i$  和  $v_j$  之间是否有边或有弧相连，则需搜索第  $i$  个或第  $j$  个链表，此操作不如邻接矩阵方便。

2. 相关类的描述

邻接表中关于边的类，可由图的边的抽象类型 `GraphEdge` 派生，同时补充指向出边的指针数据项，具体的 C++ 语言描述见程序 7-9。

程序 7-9 邻接表下的图边类描述

```
template <class TEdgeInfo>
class GraphEdgeAL:public GraphEdge <TEdgeInfo>
{
//邻接表边
private:
    GraphEdgeAL<TEdgeInfo> *next;
    //链指针，指向与本对象邻接的下一条出边
    //下面是成员函数的声明，同 GraphEdge，此处不重载，故不给出
public:
    GraphEdgeAL<TEdgeInfo>* GetNext( ) { return next; };
    void SetNext(GraphEdgeAL<TEdgeInfo> *p) { next=p; };
};
```

邻接表的图结点也可由前面介绍的图结点抽象类 `GraphNode` 派生，同时补充指向相应出边链表的头结点的指针数据项，具体的描述见程序 7-10。

程序 7-10 邻接表下的图结点类描述

```
template <class VertexType, class EdgeType>
class GraphNodeAL:public GraphNode<VertexType>
{
//邻接表结点
private:
    GraphEdgeAL<EdgeType> * firstOutEdge; //本结点的第一条出边的指针
public:
    GraphEdgeAL<EdgeType>* GetFirstOutEdge( ) { return firstOutEdge; };
    void SetFirstOutEdge(GraphEdgeAL<EdgeType>* p) { firstOutEdge=p; };
    //下面是其他成员的声明，同 GraphNode，此处不重载，故不给出
};
```

邻接表图类也是由前面定义的图的抽象类 `Graph` 派生的，为实现它的相关操作，还需

要设置指向邻接表的指针，具体的描述见程序 7-11。

程序 7-11 邻接表图类描述

```
template <class VertexType, class EdgeType>
//用到两个可变类型，分别是图结点内容和边内容
class GraphAL:public Graph<VertexType, EdgeType>
{ //邻接表类
    GraphNodeAL<VertexType, EdgeType> *nodes;
    //存放图结点的一维数组（头数组）
public:
    GraphAL( ) { nodes=NULL; numNodes=0; }; //构造函数
    GraphAL(GraphEdgeRec<EdgeType> *edgeSet, long n, long &numN)
    { EdgeSetToAL(edgeSet, n, numN); };
    GraphAL(long mNumNodes); //构造函数，只创建头数组
    GraphAL(GraphEdge<EdgeType> *edgeSet, long n);
    //此构造函数根据边集 edgeSet 创建邻接表
    ~GraphAL( ) { if (nodes) delete [ ]nodes; };
    //析构函数，负责释放邻接表内结点
    GraphNodeAL<VertexType, EdgeType> * EdgeSetToAL
    (GraphEdgeRec<EdgeType> *edgeSet, long n, long &numN);
    virtual bool FindVertexInfo(const VertexType vertexInfo)=0;
    //查找图中顶点 v 是否存在
    virtual bool LocateVertexInfo(const VertexType vertexInfo, long &pos)=0;
    //取顶点 v 在数组中的位置 pos
    virtual bool FindVertexNo(const long vertexNo);
    //找图中编号为 vertexNo 的顶点是否存在
    virtual bool LocateVertexNo(const long vertexNo, long &pos);
    //取编号为 vertexNo 的顶点在数组中的位置 pos
    virtual bool GetEdgeWeight(long v1, long v2, float &w);
    //取边<v1, v2>上的权
    virtual long GetFirstNeighbor(long v);
    //取图中顶点 v 的第一个邻接点的序号，不存在则返回-1
    virtual long GetNextNeighbor(long v);
    //取图中顶点 v 下一个邻接点的序号，不存在则返回-1
    virtual long GetNextNeighbor(long v1, long v2);
    //取图中顶点 v1 相对于 v2 的下一个邻接点的序号，不存在则返回-1
    virtual bool GetVertex(long i, GraphNode<VertexType> &node);
    //获得第 i 个结点
    virtual int SetVertex(long i, GraphNode<VertexType> *node);
    virtual bool DeleteVertex(long v); //删除编号为 v 及依附于 v 的边
    virtual bool DeleteEdge(long v1, long v2);
    //在图中删除依附于编号为 v2 和 v2 的顶点的边
    virtual long GetPrior(long vertexNo, int n);
```



```

virtual bool SetEdge(long v1, long v2, EdgeType &info, float w);
virtual long DFS2(GraphNodeAL<VertexType, EdgeType> *nodes, long n,
    long v0, char *visited, long *resu, long &top);
virtual long DFS2(long v0, long *resu);
virtual void DFSTraverse(long *resu);
virtual long DFS2_NR(int g[ ][MAX_VERTEX_NUM], long n, long v0, char
    *visited, long *resu );
virtual long DFS2_NR(long v0, long *resu);
virtual long BFS2_NR(GraphNodeAL<VertexType, EdgeType>*nodes,
    long n, long v0, char *visited, long *nos);
virtual long BFS2_NR(long v0, long *nos);
virtual void BFS2_NRTraverse(long *resu);
//...
};

```

---

### 3. 算法实现

邻接表下图的操作实现，与邻接矩阵的有所不同，下面分别进行讨论。

#### (1) 求邻接点、关联边

求以  $x$  为始点（尾）的边时，只需搜索  $x$  的出边链表（线性链表）；求以  $x$  为终点（头）的结点时，则要搜索各结点的出边链表，检查其中是否有以  $x$  为终点的链表结点，若  $i$  的线性链表中有终点为  $x$  的结点，则  $\langle i, x \rangle$  就是一条边，其中  $i$  为始点。

#### (2) 结点、边的增删

- 删除一条边：对有向图，删除边  $\langle x, y \rangle$  时，从  $x$  的出边链表中找到终点为  $y$  的结点，并删除之；对无向图，删除边  $(x, y)$  时，先在  $x$  的出边链表中找到终点为  $y$  的结点，并删除之，然后在  $y$  的出边链表中找到终点为  $x$  的结点并删除之。
- 增加新结点：在头数组中增加一个元素即可。
- 增加边：设要增加边  $\langle x, y \rangle$ ，若  $x$  与  $y$  是已存在结点，则在  $x$  的线性链表中增加一个结点，置其终点为  $y$ ；对无向图，还要按同样方法增加  $\langle y, x \rangle$ 。若  $x$  或  $y$  是新结点，则先在头数组中增加对应的元素，然后按上法将边  $\langle x, y \rangle$  加到链表中。
- 删除一个结点：是上述几项操作的复合。先删除与待删结点关联的各个边，然后再在头数组中删除该结点。

#### (3) 求度

- 出度：结点  $x$  的出度等于  $x$  的出边链表中结点个数。
- 入度：结点  $x$  的入度等于各结点的出边链表以  $x$  为终点的链结点的个数。故求入度时要扫描整个邻接表。

#### (4) 求路径

基本思想同邻接矩阵。

#### (5) 串行化问题

所谓串行化，是指如何将内存图结构转化（输出）为线性表示（表达式表示），或者如何将线性表示还原为内存结构（逆串行化），该问题对图结构的持久保存（磁盘文件保存）十分重要。对于邻接矩阵，串行化问题是数组的存储问题，所以很简单；但对邻接表，要实

现其串行化，主要需要确定结构的线性表示方式，这里不作详细介绍。

程序 7-12 给出邻接表图类的部分成员函数的实现。

程序 7-12 邻接表图类的部分成员函数

---

```
template <class VertexType, class EdgeType>
GraphAL<VertexType, EdgeType>::GraphAL(long mNumNodes)
{   numNodes=0; numEdges=0; maxNodes=mNumNodes;
    nodes=new GraphNodeAL<VertexType, EdgeType>[mNumNodes];
    for (int i=0; i<mNumNodes; i++) {   nodes[i].SetFirstOutEdge(0); }
}

template <class VertexType, class EdgeType>
bool GraphAL<VertexType, EdgeType>::FindVertexNo(const long vertexNo)
{   for (long i=0; i<numNodes; i++)
    {   if (nodes[i].GetNo()==vertexNo) return true; }
    return false;
}

template <class VertexType, class EdgeType>
bool GraphAL<VertexType, EdgeType>::GetEdgeWeight(long v1, long v2, float &w)
{   if (v1<0 || v2<0 ) return false;
    if (!FindVertexNo(v1)|| !FindVertexNo(v2)) return false;
    long pos1, pos2;
    if (LocateVertexNo(v1, pos1)&& LocateVertexNo(v2, pos2))
    {   GraphEdgeAL<EdgeType> *p; p=nodes[pos1].GetFirstOutEdge( );
        while (p!=NULL)
        {   if (p->GetEndNo()==v2) {   w=p->GetWeight( ); return true; }}
    }
    w=0;
    return false;
}

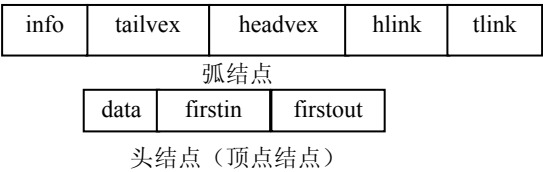
template <class VertexType, class EdgeType>
int GraphAL<VertexType, EdgeType>::SetVertex(long i,
GraphNode<VertexType> *node)
{   if (i<0)return-1;
    if (i>=numNodes && (numNodes+1)<=maxNodes)
    {   nodes[numNodes]=*((GraphNodeAL<VertexType, EdgeType>*)node);
        numNodes++; return 0;
    }
    else
    {   nodes[i]=*((GraphNodeAL<VertexType, EdgeType>*)node); return 1; }
}
```

```
template <class VertexType, class EdgeType>
bool GraphAL<VertexType, EdgeType>::SetEdge(long v1, long v2, EdgeType &info, float w)
{
    //添加一条从 v1 射向 v2 的边
    if (v1<0 || v2<0) return false;
    if (!FindVertexNo(v1)|| !FindVertexNo(v2)) return false;
    //如果 v1 或者 v2 不存在, 则返回错误信息
    long pos, pos2;
    if (LocateVertexNo(v1, pos)&& LocateVertexNo(v2, pos2))
    {
        GraphEdgeAL<EdgeType> *p, *q;
        p=nodes[pos].GetFirstOutEdge( );
        if (p==0)
        {
            numEdges++;
            p=new GraphEdgeAL<EdgeType>; p->SetNext(0); p->SetEndNo(v2);
            p->SetStartNo(v1); p->SetWeight(w); p->SetInfo(info);
            nodes[pos].SetFirstOutEdge(p);
            return true;
        }
        while (p->GetNext( )!=NULL)
        {
            if (p->GetEndNo( )==v2) break;    p=p->GetNext( );
        }
        if (p->GetEndNo( )!=v2) //如果不存在这条边, 则新增它
        {
            numEdges++;
            q=new GraphEdgeAL<EdgeType>; p->SetNext(q);
            q->SetNext(0); q->SetEndNo(v2); q->SetStartNo(v1);
            q->SetWeight(w); q->SetInfo(info);
        }
        else { p->SetWeight(w); p->SetInfo(info); } //如果存在, 则修改参数
    }
    return true;
}
...
}
```

7.3.3 十字链表（有向图）

1. 存储方法描述

十字链表（orthogonal list）是有向图的一种存储方法，它实际上是邻接表与逆邻接表的结合，即把每条弧的两个结点分别放入以弧尾顶点为头结点的链表和以弧头顶点为头结点的链表中。在十字链表中，有向图的每个弧和每个顶点都有一个结点，结点的结构如下所示：



在弧结点中有 5 个域。

- 信息域 (info): 指向该弧的相关信息。无相关信息时, 可用 “#” 表示。
- 尾域 (tailvex): 指示弧尾 (起点) 顶点编号。
- 头域 (headvex): 指示弧头 (终点) 顶点编号。
- 头链域 (hlink): 指向弧头 (终点) 相同的下一条弧。
- 尾链域 (tlink): 指向弧尾 (起点) 相同的下一条弧。

弧头相同的弧在同一链表上, 弧尾相同的弧也在同一链表上, 它们的头结点即为顶点结点, 它由 3 个域组成。

- 数据域 (data): 存储与顶点相关的信息, 如顶点的名称等。
- 入链域 (firstin): 指向以该顶点为弧头 (终点) 的第一个弧结点。
- 出链域 (firstout): 指向以该顶点为弧尾 (起点) 的第一个弧结点。

有向图的十字链表存储表示的形式说明如下所示。例如, 图 7.11 (a) 所示有向图的十字链表如图 7.11 (b) 所示。在有向图的十字链表中, 弧结点所在的链表不是循环链表, 结点之间的相对位置不一定按顶点序号排列, 表头结点即为顶点结点, 它们之间是顺序存储的。只要输入  $n$  个顶点的信息和  $e$  条弧的信息, 便可建立该有向图的十字链表。在十字链表中, 既容易找到以  $v_i$  为尾的弧, 也容易找到以  $v_i$  为头的弧, 因此容易求得顶点的出度和入度, 并且, 建立十字链表的时间复杂度与建立邻接表的是相同的。

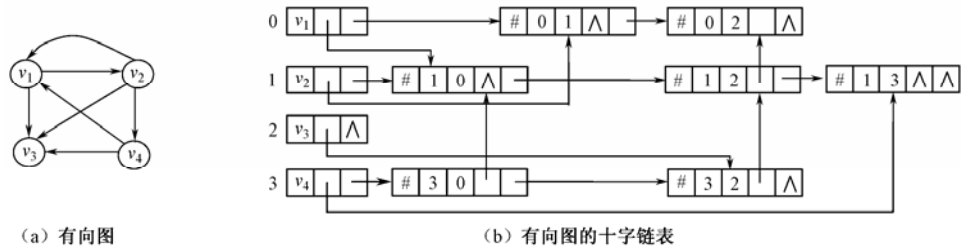


图 7.11 有向图的十字链表

2. 相关类描述

有向图的十字链表存储结构的主体是上面介绍的边结点和顶点结点, 它们也可由前面介绍的抽象边和抽象结点派生, 其中, 边结点增加了 “入链域” 和 “出链域” 的信息, 具体的 C++ 语言描述见程序 7-13。

程序 7-13 十字链表下的图边类描述

```
template <class TEdgeInfo>
class GraphEdgeCrossLink:public GraphEdge<TEdgeInfo>
{
    GraphEdge<TEdgeInfo> *tlink; //指向下一条射出边
    GraphEdge<TEdgeInfo> *hlink; //指向下一条射入边
    //下面是其他成员的声明, 同 GraphEdge, 此处不重载, 故不给出
};
```

十字链表的顶点结点也是在前面给出的抽象结点 TGraphNode 类的基础上, 增加指向相应的第一条射入边和第一条射出边的指针而成的, 具体的 C++ 语言描述见程序 7-14。

```
template <class T>
class GraphNodeCrossLink:public GraphNode<T>
{
    GraphEdge<T> *firstOut;    //指向第一条射出边
    GraphEdge<T> *firstIn;    //指向第一条射入边
    //下面是其他成员的声明，同 TGraphNode0，此处不重载，故不给出
};
```

十字链表类则是从 TGraph 派生而来的，并增加头数组作为数据成员，以便访问整个十字链表，见程序 7-15。

程序 7-15 十字链表下的图类描述

```
template <class VertexType, class EdgeType>
//用到两个可变类型，分别是图结点内容和边内容
class GraphCrossLink:public Graph<VertexType, EdgeType>
{
public:
    GraphNodeCrossLink<VertexType> nodes[MAX_VERTEX_NUM];
    //存放图结点的一维数组
    //下面是其他成员的声明，同 Graph，此处略（但实际上机时不可省略）
};
```

3. 算法实现

十字链表同时具备邻接表与逆邻接表的优点，即对任意一个结点，可容易地找出以它为始点的各边（点）和以它为终点的各边（点），但由于十字链表中每个链表结点（图中的边对应的结点）有两个链域，所以它的存储效率要比邻接表低。如果将图的邻接矩阵用矩阵的十字链表表示，可发现，图的十字链表是图的邻接矩阵的十字链表的变形。二者的主要差别是：①前者的链表结点不需按起点与终点（相当于行号与列号）的大小顺序排列；②前者的“行”与“列”链表一般不需构成循环结构。

7.3.4 邻接多重表（无向图）

邻接多重表（adjacency multilist）是存储无向图的一种链式结构。因为虽然用邻接表来存储无向图可以很容易求得图的顶点和边的各种信息，但邻接表将每条边（ $v_i, v_j$ ）的两个结点放在不同的链表中，使得某些图的操作很不方便。例如，对图中已访问过的边作记号或删除一条边等，都需要找到同一条边的两个结点。因此，当进行这一类操作时，无向图多采用邻接多重表进行存储。

邻接多重表的存储结构和十字链表类似，也是由顶点表和边表组成的，每一条边用一个结点表示，其边表结点由以下 6 个域组成：

info	mark	ivex	ilink	jvex	jlink
------	------	------	-------	------	-------

其中：

- info 为与边相关的各种信息。无相关信息时，可用“#”表示。

- mark 为标志域，可用于标记该条边是否被搜索过。
- ivex 和 jvex 为该边依附的两个顶点在图中的位置。
- ilink 指向下一条依附于顶点 ivex 的边。
- jlink 指向下一条依附于顶点 jvex 的边。

每一个顶点也用 一个结点表示，它由以下两个域组成：

data	firstedge
------	-----------

其中：

- data 域存储和该顶点相关的信息；
- firstedge 域指示第一条依附于该顶点的边。

图 7.12 所示为无向图及它的邻接多重表。

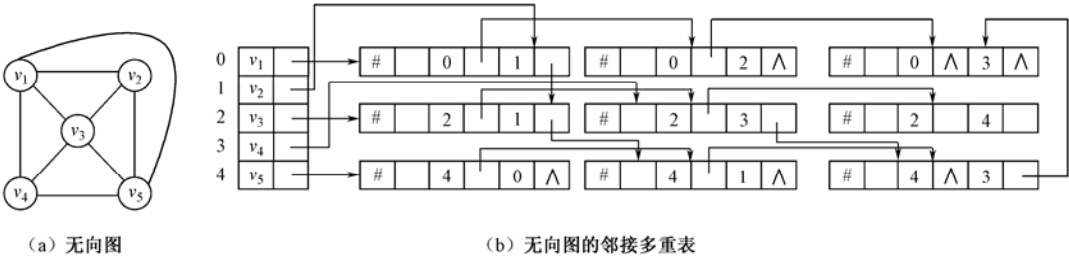


图 7.12 无向图的邻接多重表

可见，无向图两种存储方法——邻接多重表和邻接表中，它们的差别仅仅在于同一条边用两个结点表示还是用一个结点表示，因此，邻接多重表除比邻接表在边结点中多增加一个标志域外，其他方面（如各种基本操作的实现等），二者都颇为相似。

## 7.4 图的遍历

图的遍历（traversing graph）是指从某个顶点出发，对图中每个顶点各做一次且仅做一次访问，它是许多图算法的基础。由于图中任一顶点都可能和其他顶点相邻接，在访问了某顶点之后，又可能顺着某条路又回到了该顶点，也即图中可能存在回路。因此，在遍历过程中必须记住每个已访问的顶点，可设一布尔向量 `visited[0..n-1]`，其中 `visited[i]` 对应于顶点  $i$ ，若顶点  $v_i$  没有被访问，则 `visited[i]` 为 F，否则 `visited[i]` 置为 T。深度优先遍历和广度优先遍历是两种最重要的遍历图的方法，它们对无向图和有向图均适用。

### 7.4.1 深度优先遍历

#### 1. 深度优先搜索描述

无向图的深度优先搜索（depth-first search，简称 DFS）遍历是从图中某个顶点  $v$  出发，访问此顶点，然后依次从  $v$  的未被访问的邻接点出发按深度优先遍历图，直至图中所有和  $v$  有路径相通的顶点都被访问到；若此时图中尚有顶点未被访问过，则另选图中一个未曾被访问的顶点作为起始点，重复上述过程，直至图中所有顶点都被访问到为止。以图 7.13（a）中无向图  $G_4$  为例，假设从顶点  $v_1$  出发进行搜索，深度优先搜索遍历图的过程如图 7.13（b）所示。

图 7.13 (b) 中, 以带箭头的粗实线表示遍历时的访问路径, 以带箭头的虚线表示回退的路径, 小圆圈中的数字表示各顶点被访问的顺序。深度优先遍历的次序为自上而下, 从左到右。假设从  $v_1$  出发进行搜索, 在访问了顶点  $v_1$  之后, 选择邻接点  $v_2$ 。因为  $v_2$  未曾访问, 则从  $v_2$  进行搜索, 其余类推, 接着从  $v_3$  和  $v_6$  出发进行搜索。在访问了  $v_6$  之后, 由于  $v_6$  的邻接点  $v_3$  已被访问过, 则搜索回到  $v_2$ , 而  $v_2$  的另一个邻接点  $v_5$  没有被访问过, 所以从  $v_5$  开始继续搜索, 访问  $v_7$ , 之后访问  $v_7$  的邻接点, 由于  $v_5$  已被访问过, 所以搜索回到  $v_7$  的另一个邻接点  $v_8$ , 接着再搜索其邻接点  $v_9$ , 当  $v_9$  被访问过后, 并且图中所有的结点都被访问过一次, 最后得到的顶点访问序列为  $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_6 \rightarrow v_5 \rightarrow v_7 \rightarrow v_4 \rightarrow v_8 \rightarrow v_9$ 。显然, 这是一个递归的过程。

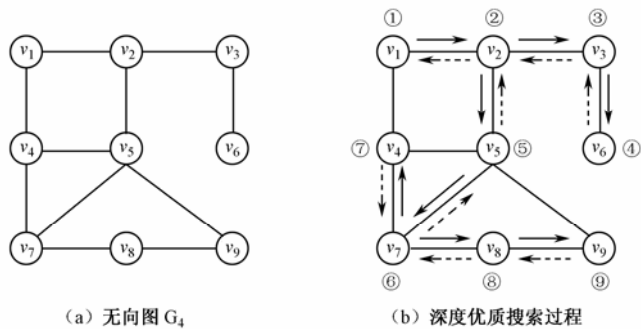


图 7.13 深度优先遍历图的过程

2. 递归算法

(1) 一般算法

下面是图的深度优先遍历递归算法的一般性描述（与存储结构无关）。

程序 7-16 深度优先遍历算法的一般性描述

```
long DFS(图 g, 结点 v0)
{
    //图深度优先遍历递归算法
    //从结点 v0 出发, 深度优先遍历图 g, 函数返回访问到的结点总数
    int nNodes;    //寄存访问到的结点数目
    访问 v0; 为 v0 置已访问标志;
    nNodes=1;
    求出 v0 的第 1 个可达邻接点 v;
    while (v 存在)
    {
        if (v 未被访问过) nNodes=nNodes+DFS(g, v);
        求出 v0 的下个可达邻接点 v;
    }
    return nNodes;
}
```

(2) 邻接矩阵下的遍历

为了简化问题, 假定图用一般的邻接矩阵存储, 邻接矩阵用简单的二维数组表示（静态）, 用 0 和 1 分别表示无边和有边, 图结点用自然数编号。

程序 7-17 给出邻接矩阵存储的图的深度优先遍历算法的实现。

```
template <class VertexType, class EdgeType>
long GraphMatrix<VertexType, EdgeType>::DFS1(int
g[ ][MAX_VERTEX_NUM], long n, long v0, char *visited, long *resu, long &top)
{//深度优先遍历图(递归)
//图 g 为邻接矩阵, 序点编号为 0~n, 函数返回实际遍历到的结点数目
//visited 是访问标志数组, 在调用本函数前, 应为其分配空间并初始化为全 0 (未访问)
//resu 为一维数组, 用于存放所遍历到的结点的编号, 在调用本函数前, 应为其分配空间
    long nNodes, i;
    nNodes=1; resu[top++]=v0; //将访问到的结点依次存于 resu[ ]中
    visited[v0]=1; //为 v0 置已访问标志
    for (i=0; i<n; i++)
        {//依次从 v0 的各个出点出发, 深度优先遍历图
            if (g[v0][i]!=0) //若<v0, i>是边
                if (!visited[i]) //若结点 i 未被访问过
                    nNodes=nNodes+DFS1(g, n, i, visited, resu, top);
                //从 i 起深度优先遍历图
        }
    return nNodes;
}
```

程序 7-17 中, 函数的参数 **visited** 和 **top** 实质上是中间变量, 只是为了避免在递归调用时重新初始化而放在参数表中, 从而造成使用的不方便。为此, 可制作包装程序 (见程序 7-18)。

#### 程序 7-18 包装程序

```
template <class VertexType, class EdgeType>
long GraphMatrix<VertexType, EdgeType>::DFS1(long v0, long *resu)
{
    char *visited;
    long top=0; visited=new char[numNodes];
    for (long i=0; i<numNodes; i++) visited[i]=0;
    long num=DFS1(smpEdges, numNodes, v0, visited, resu, top);
    delete visited;
    return num;
}
```

在上面算法中, 如果不想让 **visited** 或 **top** 作为函数参数, 也可在函数中将其定义为 **static** 类型变量。但是, 这样的程序是不可再入的, 即函数再次被调用时, **static** 类型的变量不会重新初始化, 会造成错误。程序 7-19 为使用深度优先算法遍历全部图结点的算法。

#### 程序 7-19 使用深度优先算法遍历全部图结点的算法

```
template <class VertexType, class EdgeType>
void GraphMatrix<VertexType, EdgeType>::DFSTraverse(long *resu)
```



```

{   char *visited;
    long top=0; visited=new char[numNodes];
    for (long i=0; i<numNodes; i++) visited[i]=0;
        for (i=0; i<numNodes; i++)
            if (visited[i]==0) DFS1(smpEdges, numNodes, i, visited, resu, top);
    delete visited;
}

```

---

### (3) 邻接表下的遍历

程序 7-20 给出图 g 是邻接表时深度优先遍历的实现。

程序 7-20 邻接表下的深度优先遍历算法

```

template <class VertexType, class EdgeType>
long GraphAL<VertexType, EdgeType>::DFS2(GraphNodeAL<VertexType,
    EdgeType> *nodes, long n, long v0, char *visited, long *resu, long &top)
{ //深度优先遍历用邻接表表示的图
//nodes 是邻接表的头数组, n 为结点个数(编号为 0~n), v0 为遍历的起点
//函数返回实际遍历到的结点的数目
//visited 是访问标志数组, 调用本函数前, 应为其分配空间并初始化为全 0 (未访问)
//resu 为一维数组, 用于存放所遍历到的结点的编号, 调用本函数前, 应为其分配空间
    long nNodes;
    GraphEdgeAL<EdgeType> *p;
    nNodes=1; resu[top++]=v0; //将访问到的结点依次存于 resu[ ]中
    visited[v0]=1; //为 v0 置已访问标志
    p=nodes[v0].GetFirstOutEdge(); //求出 v0 的第一个出点 p
    while (p!=NULL)
    { if (!visited[p->GetEndNo()])
        //若 p 未访问过, 则从 p 出发深度优先遍历
        nNodes=nNodes+DFS2(nodes, n, p->GetEndNo(), visited, resu, top);
        p=p->GetNext(); //令 p 指向 v0 的下个出点
    }
    return nNodes;
}

```

---

与邻接矩阵的情况类似, 也可对该程序进行“包装”, 以隐蔽 visited 和 top 参数, 程序实现留给读者自行完成。

## 3. 非递归算法

### (1) 一般算法

下面考虑深度优先遍历的非递归实现的一般方法 (与具体存储结构无关), 见程序 7-21。

程序 7-21 深度优先遍历非递归算法的一般性表述

```

long DFS_NR(图 g, 结点 v0)
{ //图深度优先遍历非递归算法

```

```

//从结点 v0 出发，深度优先遍历图 g，函数返回访问到的结点总数
int nNodes; //寄存访问到的结点数目
访问 v0; 为 v0 置已访问标志; v0 进栈 S;
nNodes=1; 求出 v0 的第 1 个可达邻接点 v;
while (栈 S 不空)
{
    v=栈 S 顶部元素;
    求 v 的下个未访问过的出点 i;
    访问 i; 为 i 置已访问标志; i 进栈 S;
    nNodes++;
    if (v 已无未被访问过的出点) 出栈;
}
return nNodes;
}

```

---

## (2) 邻接矩阵下的遍历

邻接矩阵下的深度优先遍历非递归算法见程序 7-22。

程序 7-22 邻接矩阵下的深度优先遍历非递归算法

```

template <class VertexType, class EdgeType>
long GraphMatrix<VertexType, EdgeType>::DFS1_NR(int
g[ ][MAX_VERTEX_NUM], long n, long v0, char *visited, long *resu)
{//深度优先遍历图(非递归)
//图 g 为邻接矩阵，结点编号为 0~n，返回实际遍历到的结点数目
//resu 为一维数组，用于存放所遍历到的结点的编号，调用本函数前，应为其分配空间
    long nNodes, i, v; long top; long *s;
    for (i=0; i<n; i++) visited[i]=0; s=new long[n+1]; top=0; nNodes=0;
    resu[nNodes++]=v0; //将访问到的结点依次存于 resu[ ] 中
    visited[v0]=1; //为 v0 置已访问标志
    top++; s[top]=v0;
    while (top!=0)
    {
        v=s[top];
        for (i=0; i<n; i++) //寻找 v 的下个未访问的邻接点
            if (g[v][i]!=0) //若<v, i>是边
                if (!visited[i]) //若结点 i 未被访问过
                {
                    resu[nNodes++]=i; //将访问到的结点依次存于 resu[ ] 中
                    visited[i]=1; //为 i 置已访问标志
                    top++; s[top]=i; //i 进栈
                    break;
                }
        if (i==n) top--; //若 v 已无未访问的出点，则将其退栈
    } //while
    return nNodes;
}

```

---

邻接表下的遍历算法实现，在此不给出具体程序，留作练习。

## 7.4.2 广度优先遍历

### 1. 广度优先搜索描述

广度优先搜索（breadth first search，简称 BFS）遍历从图中某顶点  $v$  出发，在访问  $v$  之后依次访问  $v$  的各个未被访问过的邻接点，然后分别从这些邻接点出发再依次访问它们的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，直至图中所有已被访问顶点的邻接点都被访问到为止。若此时图中尚有顶点未被访问过，则另选图中一个未曾被访问过的顶点作为起始点，重复上述过程，直至图中所有顶点都被访问到为止。换句话说，广度优先搜索遍历图的过程是以  $v$  为起始点，由近至远，依次访问和  $v$  有路相通且路径长度为 1, 2, ... 的顶点。

例如，对如图 7.13 (a) 所示的图  $G_4$  进行广度优先搜索遍历的过程为：首先访问  $v_1$  和  $v_1$  的邻接点  $v_2$  和  $v_4$ ，然后依次访问  $v_2$  的邻接点  $v_3$  和  $v_5$  及  $v_3$  的邻接点  $v_6$ ，接着访问  $v_5$  的邻接点  $v_7$  和  $v_9$ ，最后访问  $v_7$  的邻接点  $v_8$ 。由于这些顶点的邻接点均已被访问过，并且图中所有顶点都被访问过，由此完成了图的遍历，得到的顶点访问序列为  $v_1 \rightarrow v_4 \rightarrow v_2 \rightarrow v_5 \rightarrow v_3 \rightarrow v_6 \rightarrow v_7 \rightarrow v_9 \rightarrow v_8$ 。

### 2. 遍历算法

#### (1) 一般方法

对于深度优先遍历，用递归方法描述很自然，而广度优先遍历则不然，使用递归描述反而会使问题复杂化，所以这里只讲非递归描述法。广度优先遍历是一种分层处理，对这种分层处理，使用队列是自然的，因此设立一个队列，并保证它在任何时刻满足下列条件：

- 队中元素是已访问过的结点的可达邻接点；
- 队中元素是尚未被访问过的；
- 队中元素按它们所处的层次的先后排列。

这样，可不断地每次从队中取出一个元素并访问它，然后再将该元素的尚未被访问过的邻接点进队，直至队空。程序 7-23 给出一般的算法描述（与存储结构无关）。

程序 7-23 广度优先遍历的一般算法描述

```
long BFS(图 g, 结点 v0)
{ //在图 g 中从 v0 出发按广度优先遍历方式遍历图 g，函数返回遍历到的结点数
    long nNodes=0; 初始化队 Q;
    if (v0 存在) { v0 入队 Q; 置 v0 为已访问标志; }
    while (Q 不空)
    { 队 Q 头元素出队并送 v; 访问 v;
      nNodes++; //对已访问元素计数
      求出 v 的第一个可达邻接点 w;
      while (w 存在)
      { if (w 尚未被访问过) { w 入队 Q; 置 w 为已访问标志; }
        求 v 的下个可达邻接点 w;
      }
    }
}
```

```

    }
    return nNodes;
}
}

```

## (2) 邻接矩阵下的遍历算法

假设图用邻接矩阵表示，则它的广度优先遍历算法参见程序 7-24。

程序 7-24 邻接矩阵下的广度优先遍历算法

```

template <class VertexType, class EdgeType>
long GraphMatrix<VertexType, EdgeType>::BFS1_NR(int
g[ ][MAX_VERTEX_NUM], long n, long v0, char *visited, long *nos)
{//广度优先遍历（邻接矩阵），从 v0 出发遍历用邻接矩阵表示的图 g（共 n 个结点）
//将访问到的结点的编号存入 nos（其必须在外面分配 n 个 long 型空间）中
//函数返回遍历到的结点数目
    long nNodes=0; long v, w;
    SeqQueue<long> Q(n+1);
    if (v0>=0 && v0<n) { Q.Insert(v0); visited[v0]=1; }
    while (!Q.IsEmpty())
    {
        Q.Delete(v); nos[nNodes]= v; //访问结点 v
        nNodes++;
        for (w=0; w<n; w++) //找 v 的各未访问过的出点
            if (g[v][w]!=0)
                if (!visited[w]) { Q.Insert(w); visited[w]=1; } //v 的各未访问过的出点进栈
    } //while
    return nNodes;
}

```

这里也可设计一个对参数 **visited** 和 **top** 进行处理的包装程序，留给读者自行实现。

## (3) 邻接表下的遍历算法（见程序 7-25）

程序 7-25 邻接表下的广度优先遍历算法

```

template <class VertexType, class EdgeType>
long GraphAL<VertexType, EdgeType>::
BFS2_NR(GraphNodeAL<VertexType, EdgeType>*nodes, long n, long v0, char
*visited, long *nos)
{//广度优先遍历（邻接表），从 v0 出发遍历用邻接表表示的图 g（共 n 个结点）
//将访问到的结点的编号存入 nos（其必须在外面分配 n 个 long 型空间）中
//函数返回遍历到的结点数目
    long nNodes=0; long v;
    GraphEdgeAL<EdgeType> *p;
    SeqQueue<long> Q(n+1);
    if (v0>=0 && v0<=n) Q.Insert(v0);
    while (!Q.IsEmpty())

```

```

{   Q.Delete(v); nos[nNodes]= v;  //访问结点 v
    visited[v]=1; nNodes++; p=nodes[v].GetFirstOutEdge( );
    while (p!=NULL)
    {   if (!visited[p->GetEndNo( )])
        {   Q.Insert(p->GetEndNo( ) ); visited[p->GetEndNo( )]=1;  }
        p=p->GetNext( );
    } //while
} //while
return nNodes;
}

```

## 7.5 图的连通性问题

图的连通性问题实质上是图的遍历的一种应用。下面介绍利用图的遍历求得图的连通分量及最小代价生成树的方法。

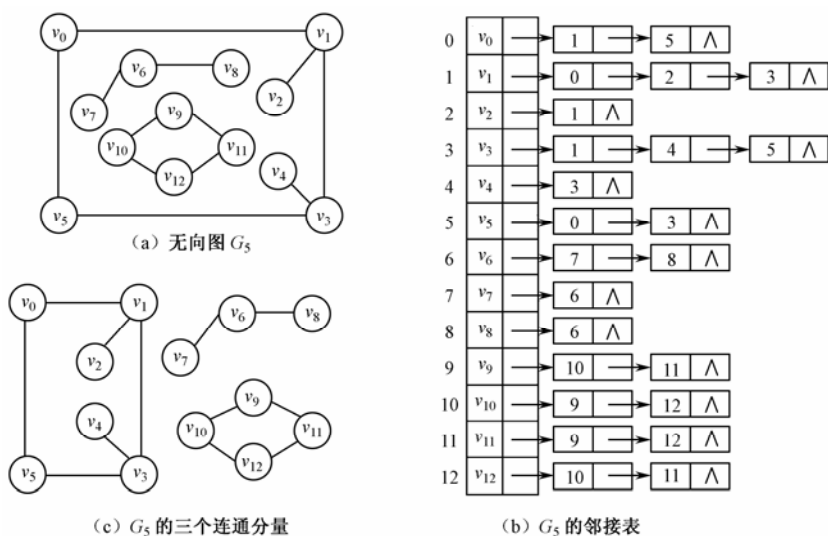


图 7.14 无向图  $G_5$ 、它的邻接表及其三个连通分量

### 7.5.1 图的连通分量

以无向图的遍历为例，若是连通图，则从图中任一顶点出发进行深度优先搜索或广度优先搜索，便可访问到图中所有顶点；若是非连通图，则需要从多个顶点出发进行搜索，由此而得到图中各个连通分量中的顶点集。例如，图 7.14 (a) 中的  $G_5$  是非连通图，按照图 7.14 (b) 所示  $G_5$  的邻接表进行深度优先搜索遍历，三次调用 DFS 过程（分别从顶点  $v_0$ 、 $v_6$  和  $v_9$  出发）得到的顶点访问序列为： $v_0 v_1 v_3 v_5 v_4 v_2$ ， $v_6 v_7 v_8$ ， $v_9 v_{10} v_{12} v_{11}$ 。这三个顶点集分别加上所有依附于这些顶点的边，便构成了非连通图  $G_5$  的三个连通分量，如图 7.14 (c) 所示。

7.5.2 生成树及生成森林

设  $E(G)$  为连通子图  $G$  中所有边的集合，则从图中任一顶点出发遍历图时，必定将  $E(G)$  分成两个集合  $T(G)$  和  $B(G)$ ，其中  $T(G)$  是遍历过程中经历的边的集合， $B(G)$  是剩余边的集合。显然， $T(G)$  和图  $G$  中所有顶点一起构成连通图  $G$  的极小连通子图，是连通图的一棵生成树，并称由深度优先搜索得到的为深度优先生成树，由广度优先搜索得到的为广度优先生成树（生成树不唯一）。例如，图 7.15 (a) 和 (b) 所示分别为图 7.13 (a) 所示  $G_4$  的深度和广度优先生成树。对于非连通图，每个连通分量中的顶点集和遍历走过的边一起构成若干棵生成树，这些连通分量的生成树组成非连通图的生成森林。图 7.15 (c) 所示为  $G_5$  的深度优先生成森林，它由三棵深度优先生成树组成。

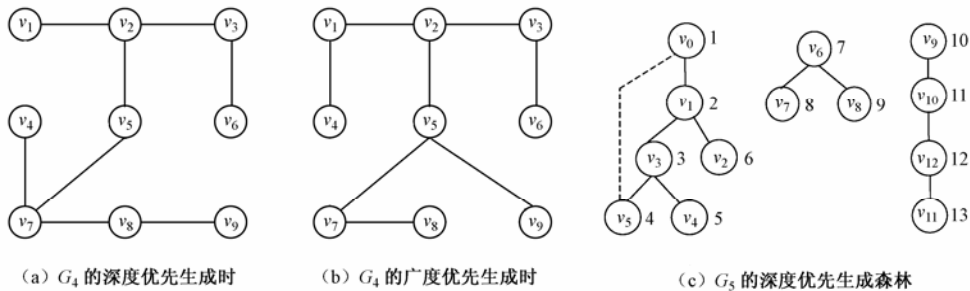


图 7.15 生成树和生成森林

程序 7-26 给出了深度优先搜索非连通无向图子连通分是建立生成树的算法，程序 7-27 给出了深度优先搜索非连通无向图建立生成森林的算法。显然，程序 7-27 的时间复杂度和遍历算法的相同。

程序 7-26 深度优先搜索非连通无向图子连通分量建立生成树的算法

```
template <class VertexType, class EdgeType>
void DFSTree(GraphMatrix<VertexType, EdgeType> &G, int v,
CSNode<VertexType>* &T, char *visited) {
//从第 v 个顶点出发深度优先遍历图 G，建立以 T 为根的生成树
    visited[v]=1; int first=1;
    CSNode<VertexType> *p, *q;
    long w;
    for (w=G.GetFirstNeighbor(v); w>=0; w=G.GetNextNeighbor(v, w))
    {   if (!visited[w])
        {   p=new CSNode<VertexType>;    //分配孩子结点
            GraphNode<VertexType> node; G.GetVertex(w, node);
            p->SetData(node.GetInfo( ));
            p->SetFirstChild(0); p->SetNextSibling(0);
            if (first) //w 是 v 的第一个未被访问的邻接顶点
            { T->SetFirstChild(p);
                first=0; //是根的左孩子结点
            } //if
        }
    }
```

```

        else { //w 是 v 的其他未被访问的邻接顶点
            q->SetNextSibling(p); //是上一邻接顶点的右兄弟结点
        } //else
        q=p; DFSTree(G, w, q, visited);
        //从第 w 个顶点出发深度优先遍历图 G，建立子生成树 q
    } //if
} //for
} //DFSTree

```

---

### 程序 7-27 深度优先搜索非连通无向图建立生成森林的算法

---

```

template <class VertexType, class EdgeType>
void DFSForest(GraphMatrix<VertexType, EdgeType> &G,
CSNode<VertexType>* &T)
{//建立无向图 G 的深度优先生成森林的（最左）孩子（右）兄弟链表 T
    T=NULL; char *visited; long v;
    CSNode<VertexType> *p, *q;
    visited=new char[G.NumOfVertexes( )];
    for (v=0; v<G.NumOfVertexes( ); ++v) visited[v]=0;
    for (v=0; v<G.NumOfVertexes( ); ++v)
        if (!visited[v]) { //第 v 顶点为新的生成树的根结点
            p=new CSNode<VertexType>; //分配孩子结点
            GraphNode<VertexType> node;
            G.GetVertex(v, node); //给该结点赋值
            p->SetData(node.GetInfo( )); p->SetFirstChild(0); p->SetNextSibling(0);
            if (!T)T=p; //是第一棵生成树的根(T 的根)
            else q->SetNextSibling(p);
            //是其他生成树的根（前一棵的根的兄弟）
            q=p; //q 指示当前生成树的根
            DFSTree(G, v, p, visited); //建立以 p 为根的生成树
        } //if
    } //DFSForest
}

```

---

**【例 7-1】** 试编写算法实现下述运算：① 建立无向图的邻接矩阵；② 建立邻接表；③ 图的深度优先搜索；④ 图的广度优先搜索；⑤ 图的连通分量计算。

---

```

const  vnum=...; //图的顶点数
struct graph
{
    int vex[vnum]; int arcs[vnum][vnum];
}; graph ga;

void buildgraph(int n, int e) //建立无向图的邻接矩阵
{
    int i, j, k, w;
    for (i=0; i<n; i++) cin>>ga.vex[i]; //读入 n 个顶点的信息

```

```

        for (i=0; i<n; i++) for (j=0; j<e; j++) //e 为边数目
            ga.arcs[i][j]=maxint; //将邻接矩阵的每个元素初始化成 maxint
                                //计算机内, 用最大整数 maxint 表示  $\infty$ 

    for (k=0; k<e; k++)
    {   cin>>i>>j>>w; ga.arcs[i][j]=w; ga.arcs[j][i]=w; } //输入边<i, j>和权
}

```

---

邻接表的表头结点形式为: **vertex link**; 表结点形式为: **adjvex next**。

---

```

struct edgenode
{   int adjvex;
    edgenode *next;
};

struct vexnode
{   int vertex;
    edgenode *link;
};

void build_adjlist(vexnode *ga[ ], int n, int e) //建立邻接表 ga, 顶点数 n, 边数 e
{   edgenode *p;
    int i, j;
    for (i=0; i<n; i++) //初始化邻接表
        {   ga[i]->vertex=i; ga[i]->link=NULL; }
        for (int k=1; k <= e; k++) {   cin>>i>>j; //读入顶点对<i, j>
            p=new edgenode; p->adjvex=j; p->next=ga[i]->link; ga[i]->link=p; }
}

void dfs(vexnode *g[ ], int v1) //从 v1 出发深度优先遍历用链接表表示的图 g
{   edgenode *p;
    bool visited[n]; //访问标记数组, n 为顶点数
    cout<<v1; visited[v1]=true; //标志 v1 访问
    p=g[v1]->link; //找 v1 的第一个邻接点
    while (p != NULL)
        {   if (!(visited[p->vertex]))dfs(g, p->vertex); p=p->next; } //回溯, 找 v1 邻接点
}

```

---

在广度优先搜索中, 若对  $x$  的访问先于  $y$ , 则对  $x$  邻接点的访问也先于对  $y$  邻接点的访问。因此, 可采用队列来暂存那些刚访问过, 但可能还有未访问的邻接点的顶点。

---

```

void bfs(vexnode *g[ ], int v1)
{ //以邻接表为存储结构的广度优先搜索。q 为队列, 假设 visited 的各分量已置为 false
    int v;
    init_linkedque(q); //设置一个空队 q
    visited[v1]=true; cout<<v1;
}

```



```

in_linkedque(q, v1); //v1 入队
while (!empty(q))
{
    v=out_linkedque(q); //出链队列
    p=g[v] ->link;
    while (p != NULL)
    {
        if (!visited[p->vertex])
        {
            visited [p->vertex]=true; cout<<p->vertex<<" ";
            in _linkedque(q, p->vertex); }
        p=p->next;
    }
}
}

void component(vexnode *g[ ], bool visited[vnum]) //计算图的连通分量
{
    int v, count;
    for (v=0; v<vnum; v++)visited[v]=false; count=0;
        for (v=0; v<vnum; v++)
            if (!(visited[v]))
            { count++; cout<<"component"<<count<<":"; dfs(g, v); cout<<endl; }
}

```

如果要遍历一个非连通图，则需多次调用 dfs 或 bfs，每次都得到一个连通分量，调用 dfs 或 bfs 的次数就是连通分量的个数。上面给出的就是以邻接表为存储结构，通过调用深度优先搜索算法实现计算连通分量的算法。

## 7.6 有向无环图及其应用

### 7.6.1 有向无环图

顾名思义，有向无环图（directed acyclic graph，简称 DAG）是指一个无环的有向图，它是一种类似于有向树的特殊有向图。如图 7.16 所示为有向树、DAG 图和有向图的例子。

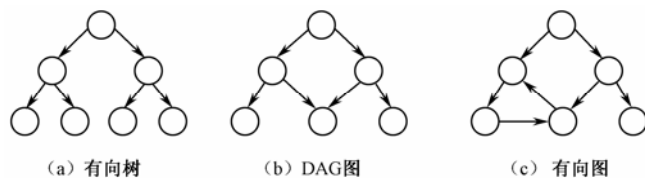


图 7.16 有向树、DAG 图和有向图示例

判断无向图是否有环，可看图的深度优先遍历过程中是否遇到指向已访问顶点的边（回边），若有，则必定存在环。对于有向图，这条回边有可能是指向深度优先生成森林中另一棵生成树上顶点的弧。但如果从有向图上某个顶点  $v$  出发进行深度优先遍历，在遍历结束之前，若出现一条从顶点  $u$  到顶点  $v$  的回边，由于  $u$  在生成树上是  $v$  的子孙，则有向图中必定

存在包含顶点  $v$  和  $u$  的环。因此，判断有向图是否存在环要比无向图复杂。

除了描述多项式以外，有向无环图也是描述一项工程或系统进行过程的有效工具。在一般情况下，工程由若干个称为“活动”（activity）的子工程组成，而这些子工程之间存在一定的约束条件（比如执行的先后次序）。对整个工程而言，工程的顺利执行以及对整个工程完成所需要的最短时间的估算是人们最为关心的两个问题，而这两个问题的解决可通过对有向图进行拓扑排序和关键路径操作来实现，下面分别进行讨论。

### 7.6.2 AOV网与拓扑排序

在一般情况下，可用有向图中的顶点表示活动，有向边表示活动之间的先后关系，这样的图被称为顶点表示活动的网（activity on vertex network, AOV 网）。在网中，若从顶点  $i$  到顶点  $j$  有一条有向路径，则  $i$  是  $j$  的前驱， $j$  是  $i$  的后继。若  $\langle i, j \rangle$  是网中一条弧，则  $i$  是  $j$  的直接前驱， $j$  是  $i$  的直接后继。AOV 网表示了活动之间的优先关系。例如，计算机专业的学生要拿到学位必须完成一系列规定的基本课程（见表 7.1），这些课程中有的独立于所有其他课程的基础课程，必须最先学习；有些课程作为某些课程的先修课程必须在那些课程之前进行学习；而有些课程可随时安排学习；等等。因此，课程的安排具有一定的先后次序。这个关系可用 AOV 网表示，如图 7.17 所示。图中，顶点表示课程，有向边（弧）表示先决条件，若课程  $i$  是课程  $j$  的先决条件，则图中有弧  $\langle i, j \rangle$ 。

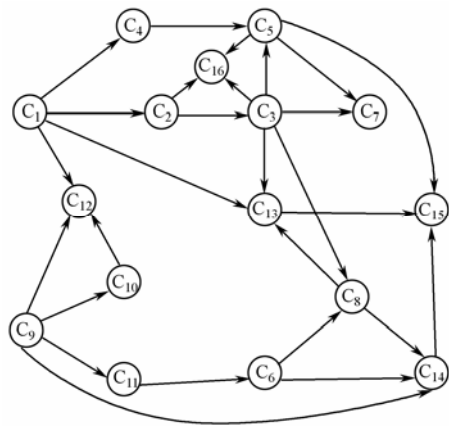


图 7.17 表示课程之间优先关系的 AOV 网

表 7.1 计算机专业课程安排

课程编号	课程名称	先决条件
C <sub>1</sub>	程序设计基础	无
C <sub>2</sub>	离散数学	C <sub>1</sub>
C <sub>3</sub>	数据结构	C <sub>1</sub> , C <sub>2</sub>
C <sub>4</sub>	汇编语言	C <sub>1</sub>
C <sub>5</sub>	算法分析	C <sub>3</sub> , C <sub>4</sub>
C <sub>6</sub>	计算机原理	C <sub>11</sub>
C <sub>7</sub>	编译原理	C <sub>3</sub> , C <sub>5</sub>

续表

课程编号	课程名称	先决条件
C <sub>8</sub>	操作系统	C <sub>3</sub> , C <sub>6</sub>
C <sub>9</sub>	高等数学	无
C <sub>10</sub>	线性代数	C <sub>9</sub>
C <sub>11</sub>	普通物理	C <sub>9</sub>
C <sub>12</sub>	计算方法	C <sub>9</sub> , C <sub>10</sub> , C <sub>1</sub>
C <sub>13</sub>	数据库原理	C <sub>1</sub> , C <sub>3</sub> , C <sub>8</sub>
C <sub>14</sub>	计算机网络	C <sub>6</sub> , C <sub>8</sub> , C <sub>9</sub>
C <sub>15</sub>	软件工程	C <sub>5</sub> , C <sub>13</sub> , C <sub>14</sub>
C <sub>16</sub>	人工智能	C <sub>2</sub> , C <sub>3</sub> , C <sub>5</sub>

为保证 AOV 网所表示的工程活动能够全部顺利执行，在 AOV 网中不应该出现回路，否则，就意味着某项活动要以自己为先决条件，显然，这是荒谬的，这样的工程无法完成。测试 AOV 网是否存在回路的办法是，对该有向图构造其顶点的拓扑有序序列，若网中所有顶点都在它的拓扑有序序列中，则该 AOV 网中必定不存在环。

所谓拓扑排序，是将有向无环图  $G$  中所有顶点排成一个线性序列，使得图中任意一对顶点  $u$  和  $v$  有以下关系：若  $\langle u, v \rangle \in E(G)$ ，则  $u$  在线性序列中出现在  $v$  之前。通常，称这样的线性序列为满足拓扑次序（topological order）的序列，简称拓扑序列。拓扑序列具有以下 5 个特点：① 若将图中顶点按拓扑次序排成一行，则图中所有的有向边均是从左指向右的；② 若图中存在有向环，则不可能使顶点满足拓扑次序；③ 一个 DAG 的拓扑序列通常表示某种方案切实可行；④ 一个 DAG 可能有多个拓扑序列；⑤ 当有向图中存在有向环时，拓扑序列不存在。

对 AOV 网进行拓扑排序的步骤是：① 在 AOV 网中选一个入度为 0 即没有前驱的顶点并输出之；② 从图中删除该顶点和所有以它为尾的弧；③ 重复上述两步，直到网中全部顶点都已输出，或者当前图中不存在入度为 0 的顶点为止。后一种情况则说明有向图中存在环。以图 7.18（a）中的 AOV 网为例，对该有向图进行拓扑排序，其过程如图 7.18（b）～（f）所示，求得其拓扑有序序列为  $v_3 \rightarrow v_1 \rightarrow v_5 \rightarrow v_2 \rightarrow v_4 \rightarrow v_6 \rightarrow v_7$ 。

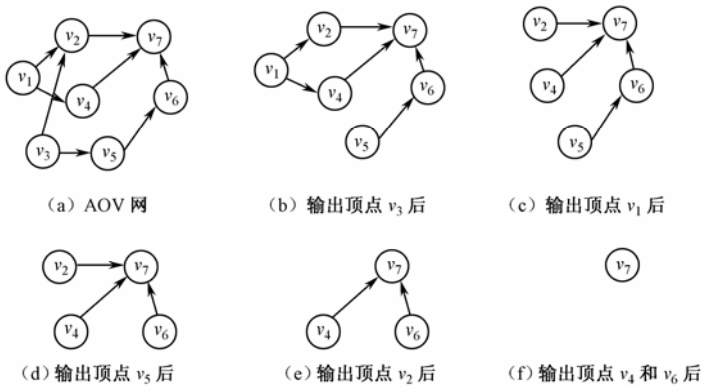


图 7.18 AOV 网及其拓扑有序序列产生的过程

对于上述操作步骤，可采用邻接表作有向图的存储结构，且在头结点中增加一个存放

顶点入度的数组 (indegree)。删除入度为零的顶点及以它为尾的弧的操作, 可通过弧头顶点的入度减 1 来实现。为避免重复检测入度为零的顶点, 可另设一栈暂存这些顶点, 由此得到拓扑排序的算法, 参见程序 7-28~程序 7-30。

#### 程序 7-28 AOV 网类

```
template <class VertexType, class EdgeType>
class GraphAOV:public GraphAL<VertexType, EdgeType>
{
protected:
    int *InDegree; //入度数组, 记录每一个顶点的入度
public:
    GraphAOV(long num=MAX_VERTEX_NUM);
    virtual ~GraphAOV() { if (InDegree) delete [ ]InDegree; };
    virtual bool SetEdge(long v1, long v2, EdgeType &info, float w);
    int *GetInDegree() const { return InDegree; }
    virtual bool TopologicalSort (int v[ ]);
};
```

#### 程序 7-29 AOV 网类主要成员函数

```
template <class VertexType, class EdgeType>
GraphAOV<VertexType, EdgeType>::GraphAOV(long num=
MAX_VERTEX_NUM):GraphAL<VertexType, EdgeType>(num)
{ InDegree=new int[num]; for (int i=0; i<num; i++) InDegree[i]=0; }
template <class VertexType, class EdgeType>
bool GraphAOV<VertexType, EdgeType>::SetEdge(long v1, long v2, EdgeType &info, float w)
{ if (v1<0 || v2<0) return false;
  if (!FindVertexNo(v1)|| !FindVertexNo(v2)) return false;
  long pos, pos2;
  if (LocateVertexNo(v1, pos)&& LocateVertexNo(v2, pos2))
  { GraphEdgeAL<EdgeType> *p, *q;
    p=nodes[pos].GetFirstOutEdge();
    if (p==0)
    { numEdges++; InDegree[pos2]++;
      p=new GraphEdgeAL<EdgeType>;
      p->SetNext(0); p->SetEndNo(v2); p->SetStartNo(v1);
      p->SetWeight(w); p->SetInfo(info);
      nodes[pos].SetFirstOutEdge(p);
      return true;
    }
  }
  while (p->GetNext() !=NULL)
  { if (p->GetEndNo() ==v2) break; p=p->GetNext(); }
  if (p->GetEndNo() !=v2) //如果不存在这条边, 则新增它
  { numEdges++; InDegree[pos2]++;
```

```

        q=new GraphEdgeAL<EdgeType>;
        p->SetNext(q); q->SetNext(0); q->SetEndNo(v2);
        q->SetStartNo(v1); q->SetWeight(w); q->SetInfo(info);
    }
    else { p->SetWeight(w); p->SetInfo(info); } //如果存在, 则修改参数
}
return true;
}

```

### 程序 7-30 拓扑排序算法

```

template <class VertexType, class EdgeType>
bool GraphAOV<VertexType, EdgeType>::TopologicalSort(int v[ ])
{
    //有向图 G 采用邻接表存储结构
    //若 G 中无回路, 则输出 G 的顶点的一个拓扑序列, 并返回 OK, 否则返回 ERROR
    SeqStack<long> S(numNodes);
    for (long i=0; i<numNodes; ++i) //建零入度顶点栈 S
        if (!InDegree[i]) S.Push(i); //入度为零者进栈
    long count, k; count=0;
    GraphEdgeAL<EdgeType> *p;
    while (!S.IsEmpty()) { S.Pop(i);
        cout<<nodes[i].GetNo( )<<" "<<nodes[i].GetInfo( )<<endl;
        //输出 i 号顶点并计数
        v[count]=i; //记录到 v[ ]中
        ++count;
        for (p=nodes[i].GetFirstOutEdge( ); p; p=p->GetNext( ))
        {
            if (!LocateVertexNo(p->GetEndNo( ), k)) return false;
            if (!(--InDegree[k])) //将 i 号顶点的每个邻接点的入度减 1
                S.Push(k); //若入度减为零, 则入栈
        } //for
    } //while
    if (count<numNodes) return false; //该有向图有回路
    else return true;
} //TopologicalSort

```

分析程序 7-30, 对有  $n$  个顶点和  $e$  条弧的有向图而言, 建立求各顶点的入度的时间复杂度为  $O(e)$ , 建零入度顶点栈的时间复杂度为  $O(n)$ 。在拓扑排序过程中, 若有向图无环, 则每个顶点进一次栈, 出一次栈, 入度减 1 的操作在 while 语句中总共执行  $e$  次, 所以总的时间复杂度为  $O(n+e)$ 。上述拓扑排序算法亦是 7.6.3 节讨论的求关键路径的基础。

#### 7.6.3 AOE网与关键路径

在带权的有向图中, 以顶点表示事件, 以有向边表示活动, 边上的权值表示活动的成本或时间等开销, 此有向图称为 AOE 网 (activity on edge network)。AOE 网与 AOV 网是既

有联系又有区别的两种图，前者用于表示一项工程中各子工程之间的优先关系，而后者用来估算工程的完成时间。通常，在 AOE 网中列出完成预定工程计划所需要进行的活动、每个活动计划完成的时间、与活动相关的事件以及事件和活动之间的关系等，由此来确定工程是否可行、总共需要的时间以及影响整个工程的关键活动。

例如，图 7.19 给出了一个有 15 项活动 ( $a_0, a_1, a_2, \dots, a_{14}$ ) 的 AOE 网，包含 10 个事件 ( $v_0, v_1, v_2, v_3, \dots, v_9$ )，其中， $v_0$  为源点，表示工程的开始， $v_9$  为汇点，表示整个工程结束。每个事件表示在它之前的活动已经完成，在它之后的活动可开始，与每个活动相联系的数是执行该活动所需的时间。在正常情况下，网中只有一个入度为零的点（源点）和一个出度为零的点（汇点）。

采用 AOE 网进行工程管理时，一般考虑两个问题：① 完成整项工程至少需要多长时间？② 哪些活动是影响工程进度的关键活动？由于 AOE 网中表示的活动可并行进行，所以完成整个工程所必须花费的最短时间是源点到汇点的最长路径长度（即路径上各活动持续时间之和），具有最大路径长度的路径称为关键路径（critical path），关键路径上的活动称为关键活动，关键路径的长度是整个工程所需的最短时间。因此，只有提前完成关键活动才能加快工程的进度，缩短整个工期。例如，图 7.19 中从  $v_0$  到  $v_9$  的最长路径是 ( $v_0, v_2, v_3, v_8, v_9$ )，路径长度是 45，也就是说，整个工程至少要 45 天才能完成。

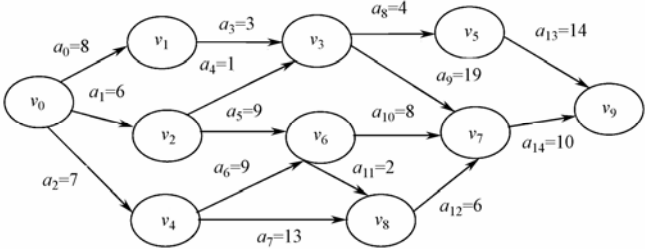


图 7.19 一个 AOE 网示例

为了找到 AOE 网中的关键路径，下面给出相关参数的定义和相应的计算方法。这里，用  $e(i)$  表示活动  $a_i$  的最早开始时间， $l(i)$  表示活动  $a_i$  的最迟开始时间，因此，关键活动就是  $e(i)=l(i)$  的活动。为了求得 AOE 网中活动的  $e(i)$  和  $l(i)$ ，先要求出事件的最早发生时间  $ve(j)$  和最迟发生时间  $vl(j)$ 。如果活动  $a_i$  用弧  $\langle j, k \rangle$  表示，其持续时间记为  $dut(\langle j, k \rangle)$ ，则有如下关系：

$$e(i)=ve(j), \quad l(i)=vl(k)-dut(\langle j, k \rangle) \tag{7-1}$$

求  $ve(j)$  和  $vl(j)$  需分两步进行：

① 从  $ve(1)=1$  开始向前递推

$$ve(j)=\text{Max}\{ve(i)+dut(\langle i, j \rangle)\} \quad \langle i, j \rangle \in T, j=1, 2, \dots, n-1 \tag{7-2}$$

式中， $T$  是所有以第  $j$  个顶点为头的弧的集合。

② 从  $vl(n-1)=ve(n-1)$  起向后递推

$$vl(i)=\text{Min}\{vl(j)-dut(\langle i, j \rangle)\} \quad \langle i, j \rangle \in S, i=n-2, n-1, \dots, 0 \tag{7-3}$$

式中， $S$  是所有以第  $i$  个顶点为尾的弧的集合。

由于  $ve(j-1)$  必须在  $v_j$  的所有前驱的最早发生时间求得之后才能确定，而  $vl(j-1)$  则必须在  $v_j$  的所有后继的最迟发生时间求得之后才能确定，因此，可在拓扑排序的基础上计算  $ve(j-1)$  和  $vl(j-1)$ ，由此得到求关键路径的算法思想如下。

① 输入  $e$  条弧  $\langle j, k \rangle$ ，建立 AOE 网的存储结构。

② 从源点  $v_1$  出发, 令  $ve[0]=0$ , 按拓扑有序求其余各顶点的最早发生时间  $ve[i]$  ( $1 \leq i \leq n-1$ )。如果得到的拓扑有序序列中顶点个数小于网中顶点数  $n$ , 则说明网中存在环, 不能求关键路径, 算法终止; 否则, 执行步骤③。

③ 从汇点  $v_n$  出发, 令  $vl[n-1] = ve[n-1]$ , 按逆拓扑有序求其余各顶点的最迟发生时间  $vl[i]$  ( $0 \leq i \leq n-2$ )。

④ 根据各顶点的  $ve$  和  $vl$  值, 求每条弧  $s$  的最早开始时间  $e(s)$  和最迟开始时间  $l(s)$ , 若某条弧满足条件  $e(s) = l(s)$ , 则为关键活动。

计算各顶点的  $ve$  值可在拓扑排序的过程中进行, 因此需对拓扑排序的算法作如下修改: ① 在拓扑排序之前设初值, 令  $ve[i] = 0$  ( $0 \leq i \leq n-1$ ); ② 在算法中增加计算  $v_j$  的直接后继  $v_k$  的最早发生时间的操作, 若  $ve(j)+dut(<j, k>) > ve[k]$ , 则  $ve[k]=ve(j)+dut(<j, k>)$ ; ③ 为了能按逆拓扑有序序列的顺序计算各顶点的  $vl$  值, 需记下在拓扑排序的过程中求得的拓扑有序序列, 这需要增设一个栈以记录拓扑有序序列, 在计算求得各顶点的  $ve$  值之后, 从栈顶至栈底便为逆拓扑有序序列。

具体算法参见程序 7-31~程序 7-33。

程序 7-31 AOE 网类

```
template <class VertexType, class EdgeType>
class GraphAOE:public GraphAOV<VertexType, EdgeType>
{
protected:
    float *ve; float *vl;
public:
    GraphAOE(int num=MAX_VERTEX_NUM);
    virtual ~GraphAOE() { if (ve)delete [ ]ve; if (vl)delete [ ]vl; };
    virtual bool TopologicalOrder(SeqStack<long> &T);
    virtual bool CriticalPath();
};
```

程序 7-32 改写的拓扑排序算法

```
template <class VertexType, class EdgeType>
bool GraphAOE<VertexType, EdgeType>::TopologicalOrder(SeqStack<long> &T)
{//有向网 G 采用邻接表存储结构, 求各顶点事件最早发生时间 ve(全局变量)
//T 为拓扑序列顶点栈, S 为零入度顶点栈
//若 G 中无回路, 则用栈 T 返回 G 的一个拓扑序列, 且函数值为 OK, 否则为 ERROR
    SeqStack<long> S(numNodes); //建零入度顶点栈 S
    long count=0;
    for (long i=0; i<numNodes; i++) ve[i]=0;
    for (i=0; i<numNodes; ++i)
    if (!InDegree[i]) S.Push(i); //入度为零者进栈
    long j, k;
    GraphEdgeAL<EdgeType> *p;
    while (!S.IsEmpty()) { S.Pop(j); T.Push(j); //j 号顶点入 T 栈并计数
```

```

++count;
for (p=nodes[j].GetFirstOutEdge( ); p; p=p->GetNext( )) {
    if (!LocateVertexNo(p->GetEndNo( ), k)) return false;
    if ((--InDegree[k])==0) //对 j 号顶点的每个邻接点的入度减 1
        S.Push(k); //若入度减为 0, 则入栈
    if (ve[j]+p->GetWeight( )> ve[k]) ve[k]=ve[j] + p->GetWeight( );
} //for *(p->info)=dut(<j, k>)
} //while
if (count<numNodes)return false; //该有向网有回路
else return true;
} //TopologicalOrder

```

---

### 程序 7-33 关键路径算法

---

```

template <class VertexType, class EdgeType>
bool GraphAOE<VertexType, EdgeType>::CriticalPath( )
{ //G 为有向网, 输出 G 的各项关键活动
    SeqStack<long> T(numNodes);
    if (!TopologicalOrder(T)) return false;
    for (long i=0; i<numNodes; i++) //初始化顶点事件的最迟发生时间
        vl[i]=ve[numNodes-1];
    GraphEdgeAL<EdgeType> *p;
    float dut, ee, el; long j, k;
    while (!T.IsEmpty( )) //按拓扑逆序求各顶点的 vl 值
        for (T.Pop(j), p= nodes[j].GetFirstOutEdge( ); p; p=p->GetNext( ))
            {
                if (!LocateVertexNo(p->GetEndNo( ), k)) return false;
                dut=p->GetWeight( ); //dut<j, k>
                if (vl[k]-dut<vl[j]) vl[j]=vl[k] - dut;
            } //for
    char tag;
    for (j=0; j<numNodes; ++j) //求 ee, el 和关键活动
        for (p=nodes[j].GetFirstOutEdge( ); p; p=p->GetNext( ))
            {
                if (!LocateVertexNo(p->GetEndNo( ), k)) return false;
                dut=p->GetWeight( ); ee=ve[j]; el=vl[k]-dut;
                tag=((ee==el)? '*': ' ');
                cout<<nodes[j].GetNo( )<<"<<nodes[k].GetNo( )<<"<<
                <<dut<<"<<ee<<"<<el<<"<<tag<<endl;
            }
    return true;
} //CriticalPath

```

---

实践证明, 用 AOE 网估算某些工程的完成时间非常有效。由于影响一个工程关键活动的因素是多方面的, 任何一项活动持续时间的改变都会造成关键路径的改变, 只有在不改变网的关键路径的情况下, 提高关键活动的速度才有效。另外, 还必须注意的是, 如果网中同



时存在几条关键路径，则整个工程工期的缩短需要提高同时在几条关键路径上的活动速度。  
 例如，对图 7.21 所示的 AOE 网的关键路径计算过程如下。

① 根据式 (7-3) 求各个事件允许的最晚发生时间。

$$\begin{aligned} vl(9) &= ve(9) = 45; \quad vl(8) = vl(9) - dut(<v_8, v_9>) = 45 - 10 = 35; \\ vl(5) &= vl(9) - dut(<v_5, v_9>) = 45 - 14 = 31; \quad vl(7) = vl(8) - dut(<v_7, v_8>) = 35 - 6 = 29; \\ vl(6) &= \min\{vl(7) - dut(<v_6, v_7>), vl(8) - dut(<v_6, v_8>)\} = \min\{27, 27\} = 27; \\ vl(3) &= \min\{vl(5) - dut(<v_3, v_5>), vl(8) - dut(<v_3, v_8>)\} = \min\{27, 16\} = 16; \\ vl(4) &= \min\{vl(6) - dut(<v_4, v_6>), vl(7) - dut(<v_4, v_7>)\} = \min\{18, 16\} = 16; \\ vl(2) &= \min\{vl(3) - dut(<v_2, v_3>), vl(6) - dut(<v_2, v_6>)\} = \min\{6, 18\} = 6; \\ vl(1) &= vl(3) - dut(<v_1, v_3>) = 13; \quad vl(0) = \min\{vl(1) - 8, vl(2) - 6, vl(4) - 7\} = \min\{5, 0, 9\} = 0. \end{aligned}$$

② 求出各个事件的 ve 和 vl 值后，再根据前面的式 (7-2) 求出各活动的最早可能开始时间  $e(i)$  和最晚发生时间  $l(i)$ ，表 7.2 所示，其中满足  $e(i)=l(i)$  的活动就是 AOE 网中的关键活动。

表 7.2 图 7.21 所示 AOE 网中顶点的发生时间和活动开始时间

顶点	ve	vl	活动	e	l	关键活动
$v_0$	0	0	$a_0$	0	5	
$v_1$	8	13	$a_1$	0	0	•
$v_2$	6	6	$a_2$	0	9	
$v_3$	16	16	$a_3$	8	13	
$v_4$	7	16	$a_4$	6	6	•
$v_5$	20	31	$a_5$	6	18	
$v_6$	16	27	$a_6$	7	18	
$v_7$	20	29	$a_7$	7	16	
$v_8$	35	35	$a_8$	16	27	
$v_9$	45	45	$a_9$	16	16	•
			$a_{10}$	16	27	
			$a_{11}$	16	27	
			$a_{12}$	20	29	
			$a_{13}$	20	31	
			$a_{14}$	35	35	•

计算结果如图 7.20 所示，可见  $a_2$ 、 $a_3$  和  $a_8$  为关键活动，组成一条从源点到汇点的关键路径。

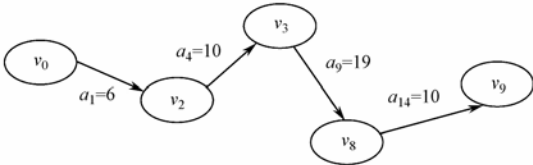


图 7.20 图 7.19 中的 AOE 网的关键路径

# 本章总结

## 1. 学习要点

本章主要介绍了图的定义及术语、图的存储结构、深度和广度优先两种遍历算法、图的连通性问题和求最小生成树及生成森林的算法、拓扑排序和关键路径求解等问题，主要学习要点如下：

- ① 图的概念和形式化定义及有关术语；
- ② 图的邻接矩阵表示法、邻接表和十字链表表示法；
- ③ 连通图遍历的基本思想和深度优先及广度优先搜索算法；
- ④ 无向图的连通分量和生成树以及有向图的强连通分量的求解方法及步骤；
- ⑤ 最小生成树的有关概念和生成算法；
- ⑥ 拓扑排序和关键路径的思想、求解步骤和算法。

## 2. 基本要求

(1) 深刻领会图的基本概念和其存储结构

- ① 知道图这种非线性数据结构的定义、术语以及与树的区别；
- ② 弄清有向图、无向图、无向网络的类型定义及其邻接矩阵和邻接表表示法的特点和区别；

③ 掌握有向图、无向图、无向网络的邻接矩阵和邻接表的建立算法；

④ 对图的十字链表和邻接多重表表示方法有一定了解。

(2) 对图的遍历应能达到“综合应用”层次

① 熟知连通图的深度优先搜索和广度优先搜索的思想、实现步骤和算法；

② 清楚非连通图的遍历方法和连通分量的求解方法；

③ 对给定的无向图，能根据它的存储结构确定其深度优先搜索和广度优先搜索输出序列。

(3) 图的连通性问题要求深刻地理解，并掌握生成树算法

① 清楚无向图的生成树、最小生成树的概念和生成算法的基本思想和执行过程；

② 能求出给定无向网络的一棵最小生成树。

(4) 对有向图及其应用问题要有深刻地理解

掌握拓扑排序及关键路径的基本思想、求解方法和步骤。

## 3. 重点和难点

重点是图形结构的概念、性质、存储方式（含邻接矩阵和邻接表）和连通图的遍历算法，难点是求图的最小生成树、拓扑排序算法和关键路径。

## 习题 7

7-1  $n$  个顶点的强连通图至少有多少条边？这样的有向图是什么形状？

7-2 对如图 7.21 所示的有向图，试给出：① 每个顶点的入度和出度；② 邻接矩阵；③ 邻接表；

④ 强连通分量；⑤ 该图是强连通的吗？

- 7-3 对如图 7.22 所示的图从顶点 1 出发，分别画出其深度和广度生成树。
- 7-4 写出分别用深度优先搜索法和广度优先搜索法遍历具有 6 个顶点的完全图的结点序列，假设都以顶点 1 为出发点。
- 7-5 写出以深度优先的搜索法的 DFS 程序过程，不用递归方法，而是使用一个栈来实现。
- 7-6 拓扑排序的结果不是唯一的，对如图 7.23 所示的拓扑序列有 52 种之多。试写出其中 10 种。

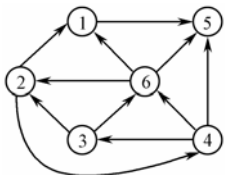


图 7.21 习题 7-2 的图

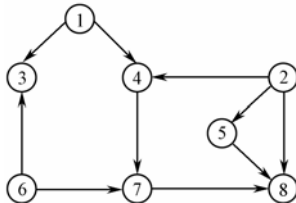


图 7.22 习题 7-3 的图

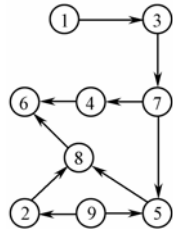


图 7.23 习题 7-6 的图

- 7-7 对于如图 7.24 所示的 AOE 网：① 求出每个活动的最早开始时间和最迟开始时间；② 完成该工程的最早时间为多少；③ 哪些活动是关键活动；④ 是否存在某些活动，只要提高其速度，就可缩短工程的时间；⑤ 画出所有可能的拓扑排序结果。
- 7-8 现把一关键 AOE 网定为其中所有活动皆为关键活动的 AOE 网，令  $G$  为该网络各边去掉指向和权之后所得到的无向图。① 证明：当且仅当  $G$  中有一条边位于自开始顶点至结束顶点的每一条路径上时，只要加速一个活动，就可使工程的长度缩短。这种边称为桥。若从一连通图中删去一座桥，便可把该图分离为两个或两个以上的连通分量。② 用邻接表写出一个  $O(e+n)$  算法，使之能确定连通图  $G$  中是否有桥。若有一座桥，应能输出它。
- 7-9 按顺序输入顶点对：(1, 2), (1, 6), (2, 6), (1, 4), (6, 4), (1, 3), (3, 4), (6, 5), (4, 5), (1, 5), (3, 5), (0, 0)，根据生成算法画出相应的邻接表。给定  $v_0 = 4$ ，按深度优先和广度优先搜索的思想编写一个算法，求一条从  $v_0$  出发到  $v_k$  点的简单路径。

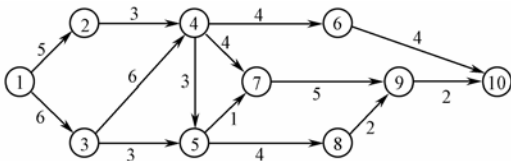


图 7.24 习题 7-7 的图

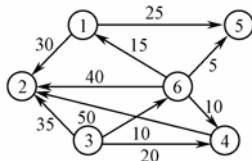


图 7.25 习题 7-10 的图

- 7-11 自己设计一个项目，画出 AOE 网，并将数据输入计算机，用程序进行分析。
- 7-12 用邻接表法存储带权的连通无向图，写出找最小生成树的算法，并估算此算法的执行时间。
- 7-13 试证明：强连通分量的归约图一定是无环有向图。
- 7-14 为了实现有向图强连通分量的算法：① 设计一种适当的数据结构，并写出由输入数据建立这种结构的过程；② 写出实现求强连通分量的过程。
- 7-15 试设计一种邻接表表示的 AOE 网，使得这种存储结构便于从起始点 1 开始，在拓扑排序的前提下，对每个顶点  $i$  求出  $ve[i]$ ；然后，再从结束点  $n$  开始，沿着有向图的反方向，在拓扑排序的前提下，对每个顶点  $i$  求出  $vl[i]$ ：① 画出这种结构的示意图；② 写一个过程，读三元组（弧的起点，终点，权）序列，建立①的存储结构；③ 按照①的结构，重写求关键路径的过程。

7-16 请为如图 7.26 所示的图设计一种表示法, 并编写一个可输入这种图并建立邻接矩阵的算法及建立邻接表的算法。

7-17 有向图(见图 7.27)是强连通图吗? 请列出所有的强连通分量, 并给出其邻接矩阵、邻接表和邻接多重表的表示。

7-18 请证明, 具有  $n$  个顶点和  $e$  条边的无向图中, 有  $\sum_{i=1}^n d_i = 2e$ , 其中  $d_i$  为顶点  $i$  的度。

7-19 设  $G$  为  $n$  个顶点的无向连通图, 则: ①  $G$  至少具有  $n-1$  条边; ② 所有具有  $n-1$  条边,  $n$  个顶点的无向连通图皆为树, 请证明之。

7-20 对具有  $n$  个顶点的无向图  $G$ , 请证明如下诸项是等价的: ①  $G$  是一棵树; ②  $G$  为连通图, 但去掉任意一条边, 即非连通图; ③ 对任何顶点  $u$  和  $v$  ( $u, v \in V(G)$ ), 恰有一条从  $u$  到  $v$  的简单路径; ④  $G$  中无回路, 有  $n-1$  条边; ⑤  $G$  是连通的, 有  $n-1$  条边。

7-21 证明: ① 用深度优先搜索一个连通图时, 遍历所经过的边将形成一棵树; ② 具有  $n$  个顶点的完全图, 两个顶点间所有路径之最大权为  $(n-1)$ 。

7-22 对于如图 7.28 所示的图: ① 用正推和逆推过程, 求出 AOE 网的每个活动的最早开始时间  $e(i)$  和最迟开始时间  $l(i)$ ; ② 该工程能完成的最早时间为多少? ③ 哪些活动是关键活动? ④ 是否存在某个活动, 加快其速度可缩短工程的时间?

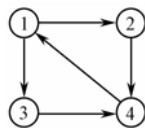


图 7.26 习题 7-16 的图

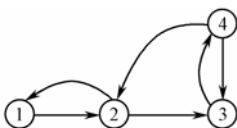


图 7.27 习题 7-17 的图

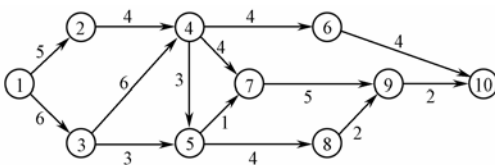


图 7.28 习题 7-22 的图

7-23 编写一套可用于对图进行操作的计算机程序。这套程序应完成如下功能: ① 对任意图进行输入、输出; ② 确定连通分量和生成树; ③ 能对各边加权, 并输出带权的图(即网)。

7-24 ① 设计一个算法, 从图的邻接矩阵构造出图的邻接表; ② 在具有  $n$  个顶点的有向图中, 把汇点定义为具有  $n-1$  条入边而没有出边的顶点, 试设计一个算法, 用以确定在  $G$  中是否已含有一个汇点(假定图的存储表示用邻接矩阵)。

7-25 设  $G=(V, E)$  是一个完全图, 即在  $G$  中的任意一对顶点之间都存在一条边。令  $G'=(V, E')$  是一个有向图。其中,  $E'$  包含  $E$  中任意指定方向的每条边, 试说明  $G'$  中存在一条有向路径, 该路径恰好经过  $V$  中的每个顶点一次。

7-26 一个无向图的欧拉回路(Euler-circuit)是这样一条路径: 从某一顶点开始, 回到同一个顶点结束, 并经过图中的每条边也恰好一次。① 试证明一个连通的无向图存在欧拉回路, 当且仅当该无向图每个顶点的度数都是偶数; ② 设计一个算法, 对给定的无向图  $G$ , 检验其是否存在欧拉回路, 如果存在, 则打印这条路径。

7-27 在无环路有向图  $G$  中, 如果从一个顶点  $v$  到其他任意顶点都有一条有向路, 则称  $v$  为  $G$  的根。试设计一个算法, 确定一个无环路有向图是否有根。

7-28 构造一个算法, 把用二元树表示的具有加 (+)、乘 (\*) 两种运算符的表达式, 转换成用无环路有向图表示, 并已包含可共享的子表达式。

# 第8章 查 找

查找运算在各种数据结构中的使用频率非常高，同时也是很多系统软件及应用程序的核心功能。当要处理的数据量非常庞大时，查找算法的效率就显得更加重要，尤其对于一些实时查询的应用。本章将系统地讨论一种广泛应用于实际问题的数据结构——查找表，主要介绍查找表的基本概念和一些基本的查找算法。

## 8.1 查找表的相关概念

### 8.1.1 基本概念

首先介绍几个基本的概念和术语。

#### (1) 关键字 (key)

数据记录中某个可标识一条记录的数据项的值称为关键字。主关键字 (primary key) 是可唯一标识一条记录的关键字，次关键字 (secondary key) 是用以识别若干条记录的关键字。

#### (2) 查找 (searching)

在一个查找表中找出关键字等于给定值  $K$  的记录。若查找成功，则返回该记录的信息或该记录在表中的位置；否则，返回相关指示信息。

#### (3) 查找表 (search table)

由同一类型的数据元素 (或记录) 构成的集合。

#### (4) 静态查找表 (static search table)

只对查找表进行“查找”操作的一类查找表。

#### (5) 动态查找表 (dynamic search table)

对查找表不仅进行“查找”操作，在查找过程中还同时插入新的数据元素或删除已有数据元素的一类查找表。

查找运算的主要操作是关键字的比较，所以通常把查找过程中对关键字需要执行的平均比较次数 (也称为平均查找长度) 作为衡量一个查找算法效率优劣的标准。平均查找长度 ASL (average search length) 定义为：

$$ASL = \sum_{i=1}^n P_i C_i \quad (8-1)$$

式中，①  $n$  是记录的个数。②  $P_i$  是查找到第  $i$  条记录的概率。若不特别声明，则认为查找每条记录的概率相等，即  $P_1=P_2=\cdots=P_n=1/n$ 。③  $C_i$  是找到第  $i$  条记录所需进行的比较次数。

### 8.1.2 类型说明

在本章后续各节中将涉及的关键字类型和数据元素类型的统一说明如下。

典型的关键字类型说明：

```
typedef float KeyType; //实型
```

```
typedef int KeyType; //整型
typedef char* KeyType; //字符串型
```

---

数据元素类型说明:

---

```
struct ElemType{
    KeyType key; //关键字域
    ... //其他域
};
```

---

## 8.2 静态查找表

### 8.2.1 概述

静态查找表的抽象数据类型定义如下。

#### ADT 8-1 静态查找表抽象数据类型

---

```
ADT StaticSearchTable {
    数据集合 D: D是具有相同特性的数据元素的集合。各个数据元素均含有类型相同。可唯一标识
        数据元素的关键字。
    数据关系 R: 数据元素同属一个集合。
    数据操作:
        ST_Create(&ST, n); //构造一个静态查找表
            输出: 构造一个含 n 个数据元素的静态查找表 ST。
        ST_Destroy(&ST); //删除一个静态查找表
            输入: 静态查找表 ST。 输出: 删除表 ST。
        ST_Search(ST, key); //在表中查找其关键字与给定值相等的记录
            输入: 静态查找表 ST 和 key, key 为与关键字类型相同的给定值。
            输出: 若 ST 中存在其关键字等于 key 的数据元素, 则函数值为该元
                素的值或在表中的位置, 否则为“空”。
        ST_Traverse(ST, visit()); //遍历静态查找表。
            输入: 静态查找表 ST, visit 是操作元素的应用函数。
            输出: 按某种次序对 ST 的每个元素调用函数 visit() 一次且仅一次。
                一旦 visit() 失败, 则操作失败。
}
```

---

### 8.2.2 顺序表的查找

顺序查找 (sequential search) 是一种最简单的查找方法, 它既适用于线性表的顺序存储结构, 也适用于链式存储结构。以顺序表或线性链表表示静态查找表, 则 Search 函数可通过顺序查找来实现。

## 1. 顺序存储结构的类型定义（见程序 8-1）

程序 8-1 顺序存储结构的类型定义

```
struct SSTable
{
    ElemType *elem;
    //数据元素存储空间基址，建表时按实际长度分配，0 号单元留空
    int length; //表长度
};
```

## 2. 顺序查找的实现

基本思想是，从表中第一个记录开始，逐个进行记录的关键字与给定值的比较，若某个记录的关键字与给定值比较相等，则查找成功；反之，若直至最后一个记录，其关键字与给定值比较都不等，则表明表中没有所查记录，查找不成功。

顺序查找算法见程序 8-20。

程序 8-2 顺序查找算法

```
int Search_Seq(SSTable &ST, KeyType key) {
    //在顺序表 ST 中顺序查找其关键字等于 key 的数据元素
    //若找到，则函数值为该元素在表中的位置；否则为 0
    int i=1;
    //从前向后查找
    while ((i<=ST.length)&&(ST.elem[i].key!=key)) i++;
    if (ST.elem[i].key==key) return i;
    else return 0;
}
```

## 3. 性能分析

(1) 只考虑查找成功的情况。在顺序查找的过程中，式 (8-1) 中的  $C_i$  取决于所查记录在表中的位置。假设  $n = \text{ST.length}$ ，则顺序查找的平均长度为  $ASL = nP_1 + (n-1)P_2 + \dots + 2P_{n-1} + P_n$ 。假设查找每个记录的概率相等，即  $P_i = 1/n$ ，则在等概率情况下顺序查找的平均查找长度为

$$ASL_{ss} = \sum_{i=1}^n P_i C_i = \frac{1}{n} \sum_{i=1}^n (n-i+1) = \frac{n+1}{2} \quad (8-2)$$

(2) 考虑查找成功和查找不成功的情况。在顺序查找的过程中，不论给定值  $key$  为何值，查找不成功时与给定值进行比较的关键字个数均为  $n+1$ 。假设查找成功与不成功的可能性相同，对每个记录的查找概率也相等，则  $P_i = 1/(2n)$ ，此时顺序查找的平均查找长度为

$$ASL_{ss} = \frac{1}{2n} \sum_{i=1}^n (n-i+1) + \frac{1}{2} (n+1) = \frac{3}{4} (n+1) \quad (8-3)$$

## 4. 顺序查找算法的特点

顺序查找的优点是，算法简单，且对表的结构无任何要求，无论用向量还是用链表来存放结点，也无论结点之间是否按关键字有序，它都同样适用。但是，顺序查找的缺点是，查找效率低，平均查找长度较大，特别是当  $n$  很大时，查找效率较低。

## 8.2.3 有序表的查找

### 1. 有序表查找的实现

以有序表标识静态查找表时，Search 函数可用折半查找来实现。有序表查找的实现如下：折半查找（binary search）是一种效率较高的查找方法，要求查找表用顺序存储结构存放且各数据元素按关键字有序（升序或降序）排列，也就是说，折半查找只适用于对有序顺序表进行查找的情况。

基本思想是，首先以整个表作为查找范围，用查找条件中给定值  $k$  与中间位置结点的关键字进行比较，若相等，则查找成功，否则，根据比较结果缩小查找范围。如果  $k$  值小于关键字的值，由表的有序性可知查找的数据元素只有可能在表的左半部分（这里假设表为升序排列），所以继续对左子表进行折半查找；若  $k$  值大于中间结点的关键字值，则可判定查找的数据元素只可能在表的右半部，所以继续对右子表进行折半查找。每进行一次折半查找，要么查找成功，结束查找，要么将查找范围缩小一半，如此重复，直到查找成功或查找范围缩小为空（即查找失败）为止。具体算法见程序 8-3。

程序 8-3 折半查找算法

```
int Search_Bin(SSTable &ST, KeyType key) {
    //在有序表 ST 中折半查找其关键字等于 key 的数据元素。
    //若找到，则函数值为该元素在表中的位置；否则为-1
    int low=0; //置区间初值
    int high=ST.length-1; //low 和 high 分别指示待查元素所在范围的下界和上界
    int middle;
    while (low <=high) {
        middle=(low + high)/2; //middle 指示区间的中间位置
        if (key>ST.elem[middle].key) low=middle+1; //找到待查元素
        else if (key<ST.elem[middle].key) high=middle-1; //继续在前半区间进行查找
        else return middle + 1; //继续在后半区间进行查找
    } //while
} //Search_Bin
```

**【例 8-1】** 已知一个有 11 个数据元素的有序表（关键字即为数据元素的值）：(04, 15, 20, 27, 39, 46, 59, 61, 78, 82, 95)，现要查找关键字为 27 和 86 的数据元素。

下面分别给出查找过程，其中的方括号表示当前的查找区间，“↑”指向中间位置、下界和上界。



Diagram illustrating the merging of sorted sub-arrays [04, 15, 20, 27, 39, 46, 59, 61, 78, 82, 95] into a single sorted array [04, 15, 20, 27, 39, 46, 59, 61, 78, 82, 95]. The process shows the sub-arrays being merged in a bottom-up fashion, with arrows indicating the flow of data from the sub-arrays to the final sorted array.

[illegible]

• 183 •

查找。下面说明在分块查找法中如何建立“索引表”。

例如，图 8.1 所示为一个表及其索引表，表中含有 18 条记录，可分成 3 块，每块包括 6 条记录，对每一个块建立一个索引项。索引表中的元素是一个结构体，包含关键字和块的起始位置，其中关键字按升序排列，即索引表为有序，各个块中的元素为有序或者分块有序。分块有序是指后一个块中的所有记录的关键字均大于前一个块中的最大关键字。

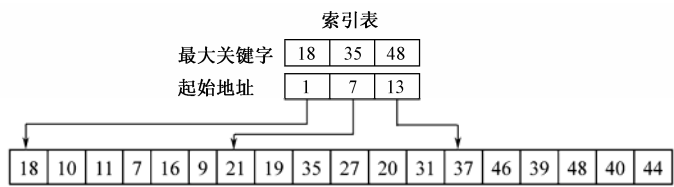


图 8.1 表及其索引表

**【例 8-2】** 试编写静态查找表的分块查找算法。

假设线性表  $F$  具有  $n$  个结点，且已按关键字排好序。把  $F$  依次划分成  $F_1$ 、 $F_2$ 、 $\cdots$ 、 $F_m$ ，共  $m$  个块。这  $m$  个块的结点个数分别为  $n_1$ 、 $n_2 \cdots n_m$ ，且  $n_1+n_2+\cdots+n_m=n$ 。为了查找关键字为  $v$  的结点，对于  $j=1, 2, \cdots, m$ ，依次将  $F_j$  的最后一个结点的关键字与  $v$  进行比较，直至  $v$  小于或等于某个  $F_j$  的最后一个结点的关键字。这时，欲找的结点落在  $F_j$  中。然后，用顺序查找法在  $F_j$  中进行查找。

具体实现时，往往把每块最大的关键字依次集中放在一个索引表中，索引表中的每个结点还带有一个指针，该指针指向线性表相应块的最后一个结点，分块查找所需的结构如图 8.2 所示。

```
const int maxn=1000; int maxm=10;
struct nodetype
{ int key, other;
};
struct indextype
{ int kindex, pindex;
};
nodetype node [maxn];
indextype index [maxm];

int blocksearch(int m, int n, int v)
{ int low, up, i, j; bool found;
  found=false; i=1;
  while ((i<=m)&&(!found))
    if (index[i-1].kinder<v) i++; else found=true;
  if (found)
  { if (i==1) low=1; else low=index[i-2].pindex+1;
    up=index[i-1].pindex; j=low; found=false;
```

```

while ((j<=up)&&(!found))
if (node[j-1].key==v) found=true; else j++;
if (found) return j; else return 0;
}
}

```

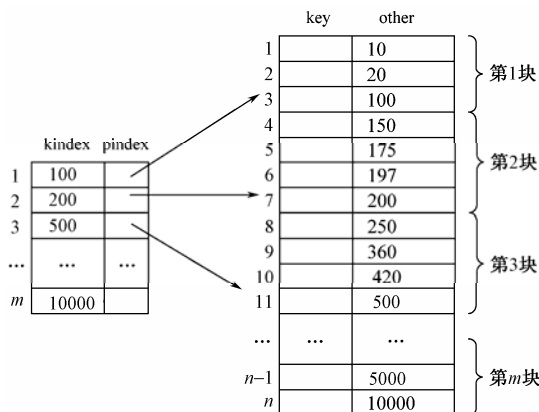


图 8.2 分块查找所需的结构

## 2. 性能分析

分块查找的平均查找长度为  $ASL_{bs} = L_b + L_w$ 。其中， $L_b$  为查找索引表确定所在块的平均查找长度， $L_w$  为在块中查找元素的平均查找长度。在一般情况下，为进行分块查找，可将长度为  $n$  的表均匀地分成  $b$  块，每块含有  $s$  个记录，即  $b = \lceil n/s \rceil$ ；又假定表中每个记录的查找概率相等，则每块查找的概率为  $1/b$ ，块中每个记录的查找概率为  $1/s$ 。分块查找的平均长度如下。

① 用顺序查找法确定所在块

$$ASL_{bs} = L_b + L_w = \frac{1}{b} \sum_{j=1}^b j + \frac{1}{s} \sum_{i=1}^s i = \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2} \left( \frac{n}{s} + s \right) + 1 \quad (8-4)$$

② 用折半查找法确定所在块

$$ASL'_{bs} \approx \log_2 \left( \frac{n}{s} + 1 \right) + \frac{s}{2} \quad (8-5)$$

## 3. 分块查找算法的特点

分块查找的优点是：① 在表中插入或删除一个记录时，只要找到该记录所属的块，就在该块内进行插入和删除运算；② 因为块内记录的存放是任意的，所以插入或删除比较容易，无须移动大量记录。分块查找的主要代价是增加了一个辅助数组的存储空间以及将初始表分块排序的运算。

# 8.3 动态查找表

## 8.3.1 概述

### 1. 动态查找表的抽象数据类型定义（见ADT8-2）

ADT8-2 动态查找表的抽象数据类型定义

ADT DynamicSearchTable {
数据集合 <i>D</i> : <i>D</i> 是具有相同特性的数据元素的集合。各个数据元素均含有类型相同、可唯一标识数据元素的关键字。
数据关系 <i>R</i> : 数据元素同属一个集合。
数据操作:
Init_DSTable(&DT); //初始化动态查找表
输出: 构造一个空的动态查找表 DT。
Destroy_DSTable(&DT); //删除动态查找表
输入: 动态查找表 DT。输出: 销毁动态查找表 DT。
Search_DSTable(DT, key); //在动态查找表中某数据元素
输入: 动态查找表 DT, key 为与关键字类型相同的给定值。
输出: 若 DT 中存在其关键字等于 key 的数据元素, 则函数值为该元素的值或在表中的位置, 否则为“空”。
Insert_DSTable(&DT, e); //在动态查找表中插入数据元素
输入: 动态查找表 DT 存在, e 为待插入的数据元素。
输出: 若 DT 中不存在其关键字等于 e.key 的元素, 则插入 e 到 DT 中。
Delete_DSTable(&DT, key); //在动态查找表中查找某个元素
输入: 动态查找表 DT 存在, key 为与关键字类型相同的给定值。
输出: 若 DT 中存在其关键字等于 key 的数据元素, 则删除之。
Traverse_DSTable(DT, visit()); //遍历动态查找表
输入: 动态查找表 DT 存在, visit 是对元素操作的应用函数。
输出: 按某种次序对 DT 的每个元素调用函数 visit() 一次且仅一次。一旦 visit() 失败, 则操作失败。
} ADT DynamicSearchTable

### 2. 动态查找表的特点

表结构本身是在查找过程中动态生成的, 即对于给定值 *key*, 若表中存在其关键字等于 *key* 的记录, 则查找成功返回, 否则插入关键字等于 *key* 的记录。

## 8.3.2 二叉排序树

### 1. 定义

二叉排序树 (binary sort tree) 又称二叉查找 (搜索) 树 (binary search tree), 是一种常

用的动态查找表。

二叉排序树可用非递归形式定义，即二叉排序树是一棵二叉树，它或者为空，或者具有如下性质：① 任一非叶子结点若有左孩子，则该结点的关键字值大于其左孩子结点的关键字值；② 任一非叶子结点若有右孩子，则该结点的关键字值小于其右孩子结点的关键字值。二叉排序树的结构如图 8.3 所示。

二叉排序树也可用递归的形式定义，即二叉排序树是一棵树，它或者为空，或者具有如下性质：① 若它的左子树非空，则其左子树所有结点的关键字值均小于其根结点的关键字值；② 若它的右子树非空，则其右子树所有结点的关键字值均大于其根结点的关键字值；③ 它的左、右子树都是二叉排序树。

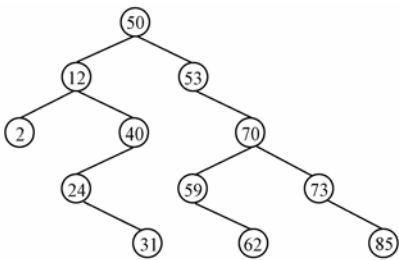


图 8.3 二叉排序树示例

## 2. 二叉排序树类的C++语言描述

由于二叉排序树一般是动态的，故采用链式结构，其结点的 C++语言描述采用前面章节介绍的二叉树结点描述，实例化时使用语句 `BSTree<ElemType> t` 定义二叉排序树对象。二叉排序树类定义见程序 8-4。

程序 8-4 二叉排序树类定义

```
template<class T>
class BSTree:public BinaryTree<T>
{
public:
    BSTree() { root=0; };
    BSTree(TBinTreeNode<T> *p)::BinaryTree(TBinTreeNode<T> *p) {};
    virtual~BSTree() {};
    bool Search(TBinTreeNode<T> *t, KeyType key, TBinTreeNode<T> *f,
                TBinTreeNode<T> * &p);
    virtual bool Insert(T e);
    bool Search(KeyType key, TBinTreeNode<T> *f, TBinTreeNode<T> *&p)
    { return Search(root, key, f, p); }
    bool DeleteBST(KeyType key) { return DeleteBST(root, key); };
    bool DeleteBST(TBinTreeNode<T> *&t, KeyType key);
    bool Delete(TBinTreeNode<T> *&p) ;
};
```

原来的部分成员函数应该重载，为了支持父类的输出操作（Output），还需要对标准输出算符进行重载，见程序 8-5。

程序 8-5 输出算符进行重载

```
ostream& operator << (ostream& oo, ElemType &il)
{ oo<<"("<<il.key<<")"; return oo; };
```

### 3. 二叉排序树的查找

算法思想是，当二叉排序树非空时，先将给定值和根结点的关键字进行比较，若相等，则查找成功，否则根据给定值和根结点的关键字之间的大小关系，分别在左子树或右子树上继续进行查找。查找算法见程序 8-6，其中结点取二叉链表作为二叉排序树的存储结构。

程序 8-6 二叉排序树查找算法

```
template<class T>
TBinTreeNode<T> * BSTree<T>::Search(TBinTreeNode<T> *t, KeyType key)
{//在根指针 T 所指二叉排序树中递归地查找某关键字等于 key 的数据元素
//若查找成功，则返回指向该数据元素结点的指针，否则返回空指针
    if ((!t) || (key==t->GetData( ).key)) return(t); //查找结束
        else if(key<(t->GetData( ).key)) //在左子树中继续查找
            return(Search(t->GetLeftChild( ), key));
        else return(Search(t->GetRightChild( ), key)); //在右子树中继续查找
} //Search
```

### 4. 二叉排序树的插入

算法思想是，根据二叉排序树的特点，新插入的结点作为叶子结点，并且在查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子，算法见程序 8-7。

程序 8-7 修改后的查找算法

```
template<class T>
bool BSTree<T>::Search(TBinTreeNode<T> *t, KeyType key, TBinTreeNode<T> *f,
TBinTreeNode<T> *&p)
{//在二叉排序树中递归地查找某关键字等于 key 的数据元素
//若查找成功，则指针 p 指向该数据元素结点，并返回 TRUE
//否则指针 p 指向查找路径上访问的最后一个结点并返回 FALSE
//指针 f 指向 T 的双亲，其初始调用值为 NULL
    if (!t) { p=f; return false; } //查找不成功
    else if EQ(key, t->GetData( ).key) { p=t; return true; } //查找成功
    else if LT(key, t->GetData( ).key) //在左子树中继续查找
        return Search((TBinTreeNode<T> *) (t->GetLeftChild( )), key, t, p);
    else return Search((TBinTreeNode<T> *) (t->GetRightChild( )), key, t, p);
        //在右子树中继续查找
} //Search
```

程序 8-8 插入算法

```
template<class T>
bool BSTree<T>::Insert(T e)
{//当二叉排序树 T 中不存在关键字等于 e.key 的数据元素时，插入 e 并返回 TRUE
//否则返回 FALSE
    TBinTreeNode<T> *s, *p;
    if (!Search(root, e.key, NULL, p)) { //查找不成功
```

```

s=new BinTreeNode<T>; s->SetData(e);
s->SetLeftChild(NULL); s->SetRightChild(NULL);
if (!p) root=s; //被插入结点*s 为新的根结点
    else if LT(e.key, p->GetData( ).key) //被插入结点*s 为左孩子
        p->SetLeftChild(s);
    else p->SetRightChild(s); //被插入结点*s 为右孩子
    return true; }
else return false; //树中已有关键字相同的结点
} //Insert

```

例如，从空树出发，经过一系列的查找插入操作之后，可生成一棵二叉树。设查找的关键字序列为{38, 24, 17, 40, 45}，则生成的二叉排序树如图 8.4 所示。

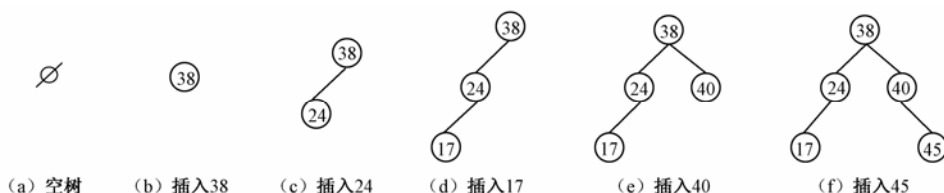


图 8.4 二叉排序树的构造过程

## 5. 二叉排序树的删除

算法思想是，在二叉排序树中删除任意一个结点时，都必须保证删除后的二叉树仍然是二叉排序树。在下列讨论过程中，假设被删结点为  $p$ ，其双亲结点为  $f$ ，删除过程如下。

① 如果  $p$  为叶子结点，则直接删除该结点；如果  $p$  为根结点，则删除后二叉排序树变为空树。

② 如果  $p$  只有左子树或只有右子树，可直接将  $f$  中指向  $p$  的指针改为指向  $p$  的左子树或右子树；如果  $p$  是  $f$  的左孩子，则  $p$  的左子树或右子树变为  $f$  的左孩子，否则  $p$  的左子树或右子树变为  $f$  的右孩子。

③ 如果  $p$  既有左子树又有右子树，根据二叉排序树的特点，可用  $p$  的中序下的前驱结点的值（或其中序下的后继结点的值）代替  $p$  的值，同时删除其中序下的前驱结点（或其中序下的后继结点），若  $p$  的中序前驱无右子树或  $p$  的中序后继无左子树，则转为②。另外，还可直接将  $p$  的右子树代替  $p$ ，同时将  $p$  的左子树变为  $p$  右子树中序第一个结点的左孩子，也可直接将  $p$  的左子树代替  $p$ ，同时将  $p$  的右子树变为  $p$  左子树中序最后一个结点的右孩子。删除过程如图 8.5 所示。

从二叉排序树中删除一个结点的算法见程序 8-9，其中由前述 3 种情况综合所得的删除算法见程序 8-10。

程序 8-9 删除一个结点算法

```

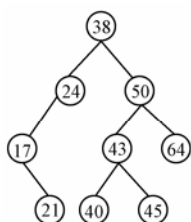
template<class T>
bool BSTree<T>::Delete(TBinTreeNode<T> *p) {
    //从二叉排序树中删除结点 p，并重接它的左子树或右子树
    TBinTreeNode<T> *q, *s;

```

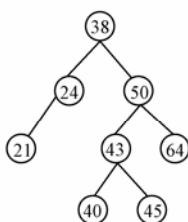
```

if (!p->GetRightChild( )) { //若右子树为空，则只需要接它的左子树
    q=p; p=p->GetLeftChild( ); delete q;
}
else if (!p->GetLeftChild( )) { //只重接它的右子树
    q=p; p=p->GetRightChild( ); delete q;
}
else { //左、右子树均不空
    q=p; s=p->GetLeftChild( );
    while (s->GetRightChild( )) { //转左，然后向右到头
        q=s; s=s->GetRightChild( );
    }
    p->SetData(s->GetData( )); //s 指向被删除结点的前驱
    if (q !=p)q->SetRightChild(s->GetLeftChild( )); //重接*q 的右子树
    else q->SetLeftChild(s->GetLeftChild( )); //重接*q 的左子树
    delete s;
} //else
return true;
} //Delete

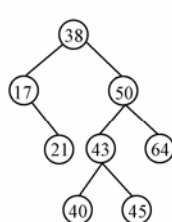
```



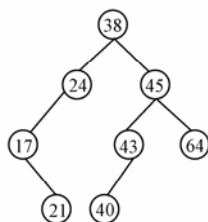
(a) 一棵二叉排序树



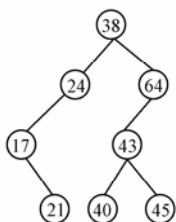
(b) 删除结点17后



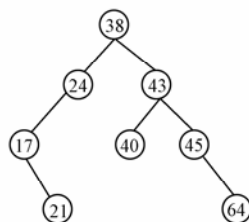
(c) 删除结点24后



(d) 删除结点50后的  
第一种情况



(e) 删除结点50后的  
第二种情况



(f) 删除结点50后的  
第三种情况

图 8.5 从二叉排序树中删除结点 p

### 程序 8-10 删除算法

```

template<class T>
bool BSTree<T>::DeleteBST(TBinTreeNode<T> *t, KeyType key)
{ //若二叉排序树 T 中存在关键字等于 key 的数据元素，则删除该数据元素结点，并返回 TRUE
  //否则返回 FALSE
  TBinTreeNode<T> *p;
  if (!t) return false; //不存在关键字等于 key 的数据元素

```



```

else
{
    if (EQ(key, t->GetData( ).key)) //找到关键字等于 key 的数据元素
    {
        return Delete(t);
    }
    else if LT(key, t->GetData( ).key)
    {
        p=t->GetLeftChild( ); bool del=DeleteBST(p, key);
        t->SetLeftChild(p); return del;
    }
    else
    {
        p=t->GetRightChild( ); bool del=DeleteBST(p, key);
        t->SetRightChild(p);
        return del;
    }
}
} //DeleteBST

```

## 6. 二叉排序树的查找分析

从以上的二叉排序树的查找算法可知，在二叉排序树中进行查找时，若查找成功，则相当于从根结点出发沿着一条路径走到待查结点；若查找失败，则相当于从根结点出发沿着一条路径走到叶子结点。类似于折半查找，二叉排序树的查找次数也不超过树的深度。由于二叉排序树是局部有序的，因此它的平均查找长度与树的形态密切相关，对于左、右子树高度相对对称的平衡二叉排序树，它的平均查找长度与折半查找相同，也是  $O(\log_2 n)$ ；但对于一棵深度为  $n$  的单枝二叉排序树，它的平均查找长度则为  $(n+1)/2$ 。但是，二叉排序树在进行插入和删除操作时无须移动大量结点，这是它相对于折半查找的优势。

### 8.3.3 平衡二叉树

#### 1. 定义和相关概念

从上面的分析可知，二叉排序树的形态直接影响了它的查找效率。树的形态越平衡越好。因此下面介绍一种平衡二叉树。平衡二叉树 (balanced binary tree, BBT) 又称 AVL 树。它或者是一棵空树，或者具有下列性质：它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过 1。

平衡因子 (balance factor, BF)：二叉树上结点的左子树深度减去它的右子树深度的值。因此，平衡二叉树上所有结点的平衡因子只可能是 -1, 0 或 1。如图 8.6 (a) 所示为一棵平衡二叉树，而如图 8.6 (b) 所示为一棵不平衡二叉树，结点中的值为该结点的平衡因子。

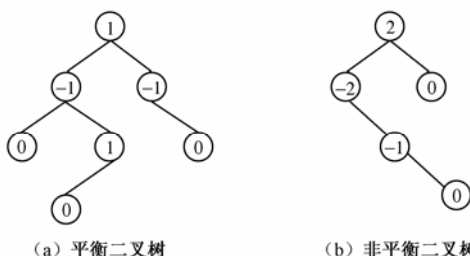


图 8.6 平衡与不平衡二叉树及结点的平衡因子

程序 8-11 给出平衡二叉树类的 C++语言描述。

程序 8-11 平衡二叉树类定义

---

```
template<class T>
class BBTreeNode:public BinTreeNode<T>
{
protected:
    int bf; //平衡因子
public:
    BBTreeNode( ) {} ;
    BBTreeNode(const T d) {} ;
    ~BBTreeNode( ) {} ;
    void SetBf(int i) { bf=i; };
    int &GetBf( ) { return bf; };
};
```

---

## 2. 平衡调整

在平衡二叉排序树中插入和删除一个结点时，通常会影响到从根结点到插入结点路径上的某些结点的平衡因子，这样也就破坏了二叉排序树的平衡性。因此，在保证二叉排序树性质的前提下，需要调整最小不平衡子树中各结点的关系以保证达到平衡。最小不平衡子树是指离插入结点最近，且插入后其平衡因子绝对值大于 1 的结点为根的子树。假设该子树的根结点为 A，则失去平衡后进行调整的规律可归纳为下列 4 种情况：① LL 型平衡处理；② RR 型平衡处理；③ LR 型平衡处理；④ RL 型平衡处理。其处理算法详见有关的参考书。

**【例 8-3】** 试编写下述算法：① 二叉排序树上的查找；② 二叉排序树的插入；③ 二叉排序树的删除；④ 平衡二叉排序树的插入。

---

```
struct btreenode
{
    int key;
    btreenode *lchild, *rchild;
}

btreenode *search_bst(btreenode *t, int k)
{//在指针 t 所指的二叉排序树上查找关键字等于 k 的结点
//成功，则回送指向结点的指针；否则回送空指针
    if (t==NULL) return NULL; //不成功，回送 NULL 作为标志
    if (t->key==k) return t; //成功，回送指针 t 作为结点
    else if (t->key > k) return search_bst(t->lchild, k); //在左子树上继续查找
    else return search_bst(t->rchild, k); //在右子树上继续查找
}

void insert_bst(int k, btreenode *t)
{
    //若 t 所指的二叉排序树上无键值为 k 的结点，则插入一个这样的结点
    btreenode *p;
    if (t==NULL)
    {
        p=new btreenode; p->key=k; p->lchild=NULL; p->rchild=NULL;
```

```

        t=p; //查找不成功，插入新结点
    }
    else
    {
        if (t->key==k) return; //查找成功，不插入
        else if (t->key > k) insert_bst(k, t->lchild) //在左子树上继续
        else if (t->key < k) insert_bst(k, t->rchild) //在右子树上继续
    }
}

void delete(btreenode *t, int a)
{
    bool found;
    btreenode *p, *q, *u, *v, *w;
    search(t, a, p, q);
    if (q !=NULL) //在树中找到结点 a, q 指向 a, p 指向 a 的前驱
    {
        if (q->lchild==NULL) v=q->rchild; //q 无左子树
        else if (q->rchild==NULL) v=q->lchild; //q 无右子树
        else //q 有左、右子树
        {
            u=q; v=q->rchild; w=v->lchild;
            while (w !=NULL) { u=v; v=w; w=w->lchild; }
            //此时, v 指向 p 按中序排序的后面结点, u 指向 v 的前面
            if (u !=q) u->lchild=v->rchild;
            v->rchild=q->rchild; v->lchild=q->lchild;
        }
        if (p==NULL) t=v else if (q==p->lchild) p->lchild=v;
        else p->rchild=v; delete q;
    }
}

```

---

### 8.3.4 B-树和B<sup>+</sup>树

#### 1. B-树

前面讨论的查找算法都是在内存中直接查找数据，适合小规模的文件，而对于规模较大的、存放在外存中的文件，如果再以结点为单位进行内存与外存设备之间的数据交换，则需要多次访问外磁盘，这样大大降低了效率。1972 年，R.Bayer 和 E.M.McCreight 提出了一种适合在磁盘等外存设备上组织动态数据的查找表，称为 B-树的多路平衡查找树。

##### (1) B-树的定义

一棵  $m$  阶的非空 B-树具有如下性质：① 树中每个结点至多有  $m$  棵子树；② 若根结点不是叶子结点，则至少有两棵子树；③ 除根之外的所有非叶子结点至少有  $\lceil m/2 \rceil$  棵子树；④ 所有的非叶子结点中包含下列信息数据：( $P_0, K_1, P_1, K_2, P_2, \dots, K_n, P_n$ )，其中， $K_i (1 \leq i \leq n)$  为关键字，且  $K_i < K_{i+1}$ ， $P_i (1 \leq i \leq n)$  为指向子树根结点的指针，且指针  $P_{i-1}$  所指子树中所有结点的关键字均小于  $K_i (i=1, 2, \dots, n)$ ， $P_n$  所指子树中所有结点的关键字均大于  $K_n$ ， $n (\lceil m/2 \rceil - 1 \leq n \leq m-1)$  为关键字的个数；⑤ 所有叶子结点都出现在同一层次上，且不包含任何信息（实际上这些结点不存在，指向这些结点的指针为空）。

一棵 4 阶的 B-树如图 8.7 所示。

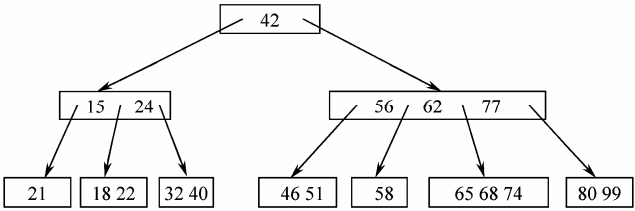


图 8.7 一棵 4 阶的 B-树

(2) B-树结点类的 C++语言描述

B-树结点类定义见程序 8-12，B-树类定义见程序 8-13。

程序 8-12 B-树结点类定义

```
class BTree;
#define m 3 //B-树的阶，暂设为3
class BTreeNode
{
    int keynum; //结点中关键字个数，即结点的大小
    BTreeNode *parent; //指向双亲结点
    KeyType key[m+1]; //关键字向量，0号单元未用
    BTreeNode *ptr[m+1]; //子树指针向量
    Record *recptr[m+1]; //记录指针向量，0号单元未用
public:
    friend BTree;
}; //B-树结点
class Result
{
    BTreeNode *pt; //指向找到的结点
    int i; //1..m, 在结点中的关键字序号
    int tag; //1:查找成功, 0:查找失败
public:
    friend BTree;
}; //B-树的查找结果类型
```

程序 8-13 B-树类定义

```
class BTree
{
    BTreeNode *root;
public:
    BTree():root(0) {}
    virtual~BTree();
    Result SearchBTree(BTreeNode *T, KeyType k);
    Result SearchBTree(KeyType k) { return SearchBTree(root, k); };
    bool InsertBTree(BTreeNode* &T, KeyType K, BTreeNode *q, int i);
    bool InsertBTree(KeyType k, BTreeNode *q, int i) { InsertBTree(root, k, q, i); }
    int Search(BTreeNode* t, KeyType k);
    bool Insert(BTreeNode* &t, int pos, KeyType x, BTreeNode* p);
};
```

### (3) B-树的查找

在 B-树上进行查找的过程是一个顺时针查找结点和在结点的关键字中进行查找交叉进行的过程。

在 B-树上进行查找包含两种基本操作：在 B-树中找结点（操作在磁盘上进行）和在结点中找关键字（操作在内存中进行）。由于在磁盘中进行查找比在内存中进行查找要耗费时间得多，因此在磁盘中的查找次数（即待查关键字所在结点在 B-树上的层数）对 B-树查找效率起主要决定作用。根据 B-树的定义，第一层至少有 1 个结点；第二层至少有 2 个结点；由于除根之外的每个非终端结点至少有  $\lceil m/2 \rceil$  棵子树，则第三层至少有  $2(\lceil m/2 \rceil)$  个结点；依次类推，第  $n+1$  层至少有  $2(\lceil m/2 \rceil)^{n-1}$  个结点，而  $n+1$  层为叶子结点。若  $m$  阶 B-树中具有  $N$  个关键字，则查找不成功的结点（即叶子结点）为  $N+1$ ，由此有  $N+1 \geq 2 \times (\lceil m/2 \rceil)^{n-1}$ ，推得  $n \leq \log_{\lceil m/2 \rceil} [(N+1)/2] + 1$ 。这说明在含有  $N$  个关键字的 B-树上进行查找时，从根结点到关键字所在结点的路径包含的结点数不超过  $\log_{\lceil m/2 \rceil} [(N+1)/2] + 1$ 。

### (4) B-树的插入

由于 B-树结点中的关键字个数必须大于等于  $\lceil m/2 \rceil - 1$ ，因此，插入关键字的过程是查找待插入的最低层的某个非叶子结点，然后把关键字添加进去，具体分为两种情况。① 若该结点的关键字个数不超过  $m-1$ ，则直接插入即可，在如图 8.8 (a) 所示的 5 阶 B-树中插入 23 后的 B-树如图 8.8 (b) 所示。② 若该结点的关键字个数大于  $m-1$ ，则要进行结点的“分裂”，即将该结点分裂为两个，并把中间的一个关键字取出来放到该结点双亲结点中去，若双亲结点中的关键字个数也大于  $m-1$ ，则需要再分裂，如果一直分裂到根结点，则需要建立一个新的结点，并且将整个 B-树层次加 1。在如图 8.8 (a) 所示的 5 阶 B-树中插入 23 后的 B-树如图 8.8 (c) 所示。

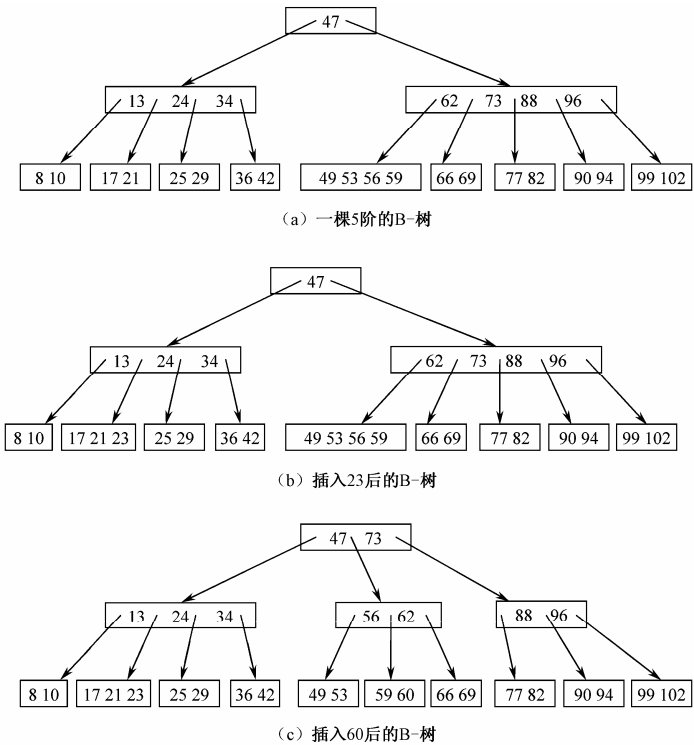


图 8.8 一棵 5 阶 B-树的插入过程

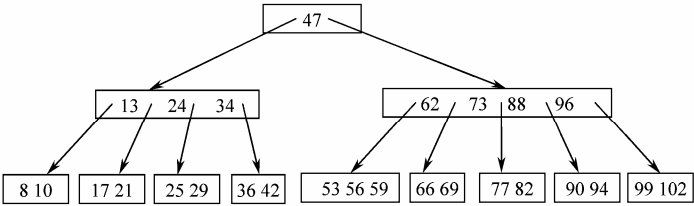
(5) B-树的删除

算法思想是，在 B-树中删除一个关键字  $K_i$ ，首先应找到该关键字所在结点，并从中删除之，删除操作分以下情况进行。

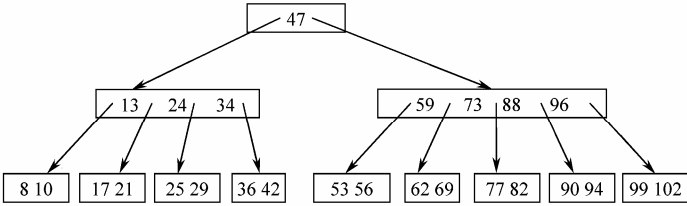
若所删关键字在底层的非叶子结点中，则有下列 3 种情况。

① 若被删关键字所在结点中的关键字数目不小于  $\lceil m/2 \rceil$ ，则只需从该结点中删去该关键字  $K_i$  和相应指针  $P_i$ ，树的其他部分不变，在如图 8.8 (a) 所示的 5 阶 B-树中删除 49 后的 B-树如图 8.9 (a) 所示。

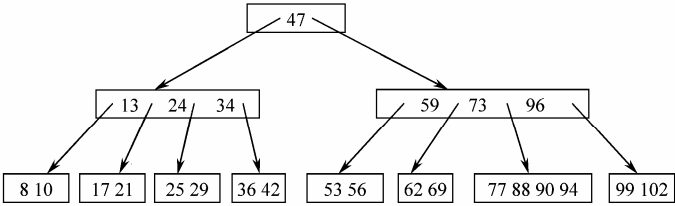
② 若被删关键字所在结点中的关键字数目等于  $\lceil m/2 \rceil - 1$ ，而与该结点相邻的右兄弟（或左兄弟）结点中的关键字数目大于  $\lceil m/2 \rceil - 1$ ，则需将其兄弟结点中的最小（或最大）的关键字上移至双亲结点中，而将双亲结点中小于（或大于）且紧靠该上移关键字的关键字下移至被删关键字所在结点中，在如图 8.9 (a) 所示的 5 阶 B-树中删除 66 后的 B-树如图 8.9 (b) 所示。



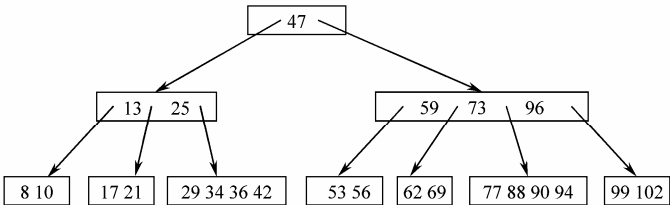
(a) 在图 8.8 (a) 中删除 49 后的 B-树



(b) 在图 8.9 (a) 中删除 66 后的 B-树



(c) 在图 8.9 (b) 中删除 82 后的 B-树



(d) 在图 8.9 (c) 中删除 24 后的 B-树

图 8.9 在 B-树中删除关键字的情形

③ 若被删关键字所在结点和其相邻的兄弟结点中的关键字数目均等于 $\lceil m/2 \rceil - 1$ ，假设该结点有右兄弟，且其右兄弟结点地址由双亲结点中的指针  $P_i$  所指，则在删去关键字之后，它所在结点中剩余的关键字和指针，加上双亲结点中的关键字  $K_i$  一起，合并到  $P_i$  所指兄弟结点中（若没有右兄弟，则合并至左兄弟结点中）。如果因此使双亲结点中的关键字数目小于 $\lceil m/2 \rceil - 1$ ，则做相应处理。在如图 8.9 (b) 所示的 5 阶 B-树中删除 82 后的 B-树如图 8.9 (c) 所示。

若所删关键字处于非最低层的分支结点中，则可用指针  $P_i$  所指子树中的最小关键字  $X$  替代  $K_i$ ，然后再按照以上 3 种情况在相应的结点中删去  $K_i$ 。在如图 8.9 (c) 所示的 5 阶 B-树中删除 24 后的 B-树如图 8.9 (d) 所示。

## 2. B<sup>+</sup>树

### (1) 定义

B<sup>+</sup>树是 B-树的一种变形，它和 B-树的差别在于：① 有  $n$  棵子树的结点中含有  $n$  个关键字；② 所有的叶子结点中包含了全部关键字的信息及指向含这些关键字记录的指针，且叶子结点本身按关键字的大小顺序链接；③ 所有的分支结点可看成是索引部分，结点中仅含其子树（根结点）中最大（或最小）的关键字。一棵 3 阶的 B<sup>+</sup>树如图 8.10 所示。通常，在 B<sup>+</sup>树上有两个指针，一个指向根结点，一个指向关键字最小的叶子结点。

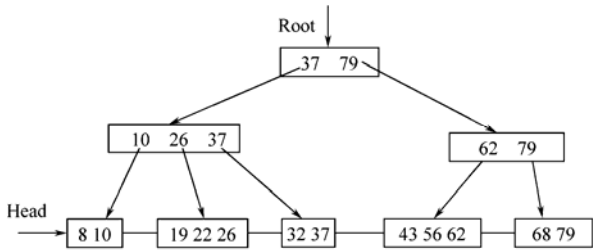


图 8.10 一棵 3 阶的 B<sup>+</sup>树

### (2) B<sup>+</sup>树的查找

对 B<sup>+</sup>树可进行两种查找运算：一种是利用 head 指针从最小关键字开始顺序查找，另一种是利用 root 指针从根结点开始进行随机查找。在查找过程中，若非叶子结点上的关键字等于给定值，此时并不结束查找，而是继续向下直到叶子结点，其余的操作同 B-树的查找类似。

### (3) B<sup>+</sup>树的插入

B<sup>+</sup>树的插入仅在叶子结点上进行，当插入后使得结点中的关键字个数大于  $m$  时，要分裂成两个结点，它们所含关键字的个数分别为 $\lceil (m+1)/2 \rceil$ 和 $\lceil m/2 \rceil$ ，并把分裂后的两个结点中的最大关键字放入它们的双亲结点中，其余的操作同 B-树的插入类似。

### (4) B<sup>+</sup>树的删除

B<sup>+</sup>树的删除也仅在叶子结点中进行，当叶子结点中的最大关键字被删除时，其在非叶子结点中的值可作为一个“分界关键字”存在，若因删除而使结点中关键字的个数小于 $\lceil m/2 \rceil$ ，则需要和兄弟结点进行合并，合并的过程与 B-树类似。

# 8.4 哈 希 表

前面介绍的各种查找方法都是将结点的关键字与待查找的值按照某种方式进行一一比较而实现的，查找的效率依赖于比较的次数。下面介绍一种通过将关键字的值与其存储位置之间建立某种映射关系而直接获得地址的方法，这就是哈希（Hash）表法。哈希表查找方法又称为散列法，其基本思想是根据结点的关键字的值来确定其存储位置，下面给出相关定义。

## 8.4.1 哈希表的定义

哈希函数：将结点的关键字与它的存储位置之间建立一种对应关系的函数称为哈希函数。

哈希表：根据设定的哈希函数  $h(key)$  和处理冲突的方法将一组关键字映射到一个有限的连续地址区间上，这样的表称为哈希表，这一映射过程称为哈希造表或散列，所得的存储位置称为哈希地址或散列地址。

冲突（collision）：不同的关键字映射到同一哈希地址的现象称为冲突。

同义词（synonym）：在一个哈希函数中，具有相同函数值的关键字，互称为同义词。

例如，存储一张学生信息表，其中每条记录包含编号、姓名、性别、年龄、民族等信息，如下所示：

num	name	sex	age	nation	...
-----	------	-----	-----	--------	-----

该表可用一个一维数组  $A$  来存放，如果将编号为  $i$  的学生记录存放到  $A[i]$  中，则编号  $num$  便为记录的关键字，由它唯一确定记录的存储位置  $A[i]$ 。假设学生张明的编号为 1，若要查看张明的个人信息，只要取出  $A[1]$  中记录即可，这个数组则可看成哈希表，哈希函数为  $h(num)=num$ ；如果取姓名为关键字，并将姓名的第一个字母在字母表中的序号作为哈希函数，则 ZHANG 的哈希函数值为字母“Z”在字母表中的序号等于 26，但会出现这种情况，即关键字 ZHEN 和 ZHANG 不相等，但  $h(ZHEN)=h(ZHANG)$ ，这说明出现了冲突。ZHEN 和 ZHANG 在该哈希函数下互称为同义词。

## 8.4.2 哈希函数的构造

建立哈希表的关键是构造哈希函数，其原则是尽可能地使任意一组关键字的哈希地址均匀地分布在整个地址空间中，即用任意关键字作为哈希函数的自变量时，使得计算结果随机分布，以减少冲突的发生。通常，构造哈希函数主要考虑如下因素：①计算哈希函数所需的时间；②关键字的长度；③哈希表的大小；④关键字的分布情况；⑤记录的查找频率。下面介绍几种常用的哈希函数构造方法。

### （1）直接地址法

取关键字或关键字的线性函数值为哈希地址，即  $h(key)=key$  或  $h(key)=a*key+b$ ，其中  $a$  和  $b$  为常数。例如，有一个包含 100 名员工信息的表格，表中每个记录包含编号、姓名、所属部门、职称等信息，若取编号作为关键字，则可利用直接地址法确定各记录的哈希存储地址。直接地址法的地址空间的大小与关键字值域的大小相同，对于不同的关键字，不会发生冲突。但由于在实际应用中，关键字往往是离散的，因此该方法会导致大量空间的浪费。



(2) 平方取中法

取关键字平方后的中间几位为哈希地址，所取的位数由哈希表的表长决定。由于一个关键字平方后所得的中间几位与该关键字的每一位都有关，因此能够使哈希地址的分布更为均匀，这种方法较为常用。

(3) 数字分析法

对于关键字的位数比存储区域的地址码位数多的情况，可在对关键字的各位进行分析后，丢掉分布不均匀的，留下分布均匀的位作为哈希地址。例如，有一组 8 位十进制数的关键字表，如下所示。假设哈希地址是 3 位的，需经过分析丢掉 5 位，这 3 位的选取要使得到的哈希地址尽量避免产生冲突。

...

6	0	3	4	6	5	3	1
6	0	5	7	2	5	4	8
6	0	3	8	7	8	2	1
6	0	3	0	1	5	6	1
6	0	6	2	2	8	1	8
6	0	3	3	8	5	6	1
6	0	3	5	4	1	5	8
6	0	5	6	8	5	3	8

...

对这一组关键字从最高位（最左边）开始进行分析可知，第 1、2 位都是 6 和 0，第 3 位有 5 个 3，第 6 位有 5 个 5，而第 8 位只有 1 和 8，所有这些都不可取，而中间的第 4、5、7 位可看成是近乎随机的，因此可取它们的组合作为哈希地址。

(4) 折叠法 (folding)

将关键字分割成位数相同的几部分（最后一部分的位数可不同），然后取这几部分的叠加和（舍去进位）作为哈希地址。折叠法又可分为移位叠加和间界叠加，其中移位叠加是将分割后的每一部分的最低位对齐，然后相加；而间界叠加是从一端向另一端沿分割界来回折叠，然后对齐相加。这种方法主要用于关键字位数很多，而且关键字中每一位数字分布大致均匀的情况。例如，关键字为  $key=84731562$ ，哈希表的长度为 1000 时，可把关键字分为 84、731、562 三个部分并进行叠加，其哈希地址分别如图 8.11 (a) 和 (b) 所示。

562	562
731	137
+ 084	+ 084
-----	-----
1377	783
H(key) = 377	H(key) = 783
(a) 移位叠加	(b) 间界叠加

图 8.11 由折叠法求得哈希地址

(5) 除留余数法

取关键字被某个不大于哈希表表长  $m$  的数  $p$  除后所得余数为哈希地址，即  $h(key) = key \text{ MOD } p, p \leq m$ 。一般，可选  $p$  为质数或不包含小于 20 的质因数的合数。例如，对于下面一组关键字，当哈希表的长度为 20 时，选  $p=19$  比较合适，哈希地址利用哈希函数  $h(k)=k\%19$  来计算，求得的结果如下所示：

关键字	39	124	83	234	195	79
哈希地址	1	10	7	6	5	3

(6) 随机数法

选择一个随机函数，取关键字的随机函数值为它的哈希地址，即  $h(\text{key})=\text{random}(\text{key})$ ，其中 **random** 为随机函数。该方法适合于关键字长度不相等的情况。

8.4.3 处理冲突的方法

冲突是指由关键字得到的哈希地址为  $j$  ( $0\leq j\leq n-1$ ) 的位置上已存在记录。冲突解决的方法是，为产生冲突的关键字的记录找到另一个“空”的哈希地址。在构造哈希函数时，很难避免冲突，因此下面将介绍几种常用的冲突解决方法。

(1) 开放定址法

开放定址法的基本思想是，当发生冲突时，按照某种方法继续探测基本表中的其他存储单元，直到找到一个空闲位置为止。它的一般形式可表示为  $h_i=(h(k)+ d_i) \text{ MOD } m, i = 1, 2, \cdots, k$  ( $k\leq m-1$ )，其中  $h(k)$ 为关键字  $k$  的直接哈希地址， $m$  为哈希表长， $d_i$  为每次再探测时的增量序列。当  $d_i=1, 2, 3, \cdots, m-1$  时，称为线性探测再散列；当  $d_i=1^2, -1^2, 2^2, -2^2, 3^2, \cdots, \pm k^2$  ( $k\leq m/2$ ) 时，称为二次探测再散列；当  $d_i$ 为伪随机数序列时，称为伪随机探测再散列。

当冲突处理过程中出现两个第一个哈希地址不同的关键字争夺同一个后继哈希地址的现象时，称为二次聚集，即在处理同义词的冲突过程中又添加了非同义词的冲突。例如，在一个长度为 13 的哈希表中，根据哈希函数  $h(\text{key}) = \text{key} \text{ MOD } 13$  存储了关键字分别为 19，28 和 42 的记录，如图 8.12 (a) 所示。现要存储第 4 个关键字 66，由哈希函数得到其哈希地址为 1，产生冲突；线性探测再散列得到下一个地址 2，仍有冲突；再探测得到下一个地址 3，仍有冲突；直到哈希地址为 4 的位置为空时，才可存入而结束探测，如图 8.12 (b) 所示。如果采用二次探测再散列方法，则将 66 填入序号为 0 的位置，如图 8.12 (c) 所示。如果采用伪随机探测再散列，伪随机数列为 5，则将 66 填入序号为 6 的位置，如图 8.12 (d) 所示。



图 8.12 用开放定址处理冲突时，关键字为 66 的记录插入前后的哈希表

从上面的过程可看出，线性探测再散列的方法是，当哈希函数产生的地址已经被占用（也就是发生冲突）时，便会到下一个地址进行寻找，即当表中  $i, i+1, i+2$  位置上都存有关键字后，下一次哈希地址为  $i, i+1, i+2, i+3$  的关键字都将试图存入  $i+3$  位置，这样就产生了“二次聚集”，显然不利于查找。但线性探测再散列方法的优点是，能充分利用表空间，即

表中的所有存储单元都可被探测并使用到。而二次探测再散列方法只有在哈希表长  $m$  为形如  $4j+3$  ( $j$  为整数) 的素数时, 才能够使得表中所有单元被探测到。至于伪随机探测再散列方法, 其表空间的利用程度取决于伪随机探测序列。

(2) 再哈希法

该方法的哈希函数为  $h_i = Rh_i(\text{key})$ ,  $i = 1, 2, \cdots, k$ , 其中  $R$ 、 $h_i$  均是不同的哈希函数。即在同义词产生地址冲突时计算另一个哈希函数地址, 直到冲突不再发生。这种方法不易产生“二次聚集”, 但增加了计算的时间。

(3) 链地址法

该方法解决冲突的处理是, 将所有关键字为同义词的记录存储在同一线性链表中。若哈希函数产生的哈希地址长度为  $n$ , 分布在区间  $[0, m-1]$  上, 则可将散列表定义为一个指针型向量  $\text{Chain\_Hash}[m]$ , 其中每个分量的初始状态为空, 哈希地址为  $i$  的关键字都链接到头指针为  $\text{Chain\_Hash}[i]$  的链表中。

该方法的优点是: 处理简单, 无“二次聚集”产生; 插入和删除操作易于实现; 链表结点可动态申请, 因此哈希表的长度不受限制; 等等。但它也具有需要额外的空间存放指针的缺点。例如, 已知一组关键字(17, 19, 36, 28, 11, 75, 83, 22, 43), 则按哈希函数  $h(\text{key}) = \text{key} \bmod 7$  和链地址法处理冲突构造所得的哈希表如图 8.13 所示。

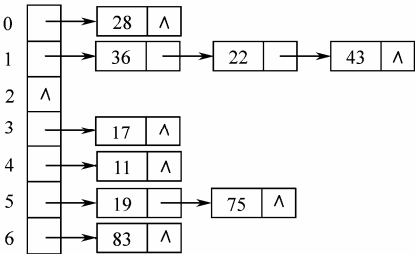


图 8.13 链地址法处理冲突时的哈希表

(4) 建立一个公共溢出区

设哈希表的地址空间为向量  $\text{HT}[0..m-1]$ , 每个分量存放一个记录, 另外再设立一个向量  $\text{OverTable}[0..v]$  为溢出表。若关键字和已存入哈希表中关键字为同义词, 则不管它们由哈希函数得到的哈希地址是什么, 一旦发生冲突, 都填入溢出表。

【例 8-4】 编写程序实现避免冲突。

为了能够检测冲突的发生, 在建立 Hash 表之前, 应对 Hash 表进行初始化。可以用不出现在记录中的符号放在 Hash 表的所有位置上, 表示 Hash 表中所有位置都是空的。当一个记录的键值用所挑选的 Hash 函数计算出它的 Hash 地址时, 如果该位置空, 则可把记录存到此位置中; 如果该位置已被其他记录抢先占用了(发生了冲突), 此时, 必须为它在 Hash 表内部寻找一个空位置, 然后再把新进入的记录存放在到刚找到的空位置上, 最简便的办法是在邻近的位置上寻找空位置。

当往下找空位置时, 若找到 Hash 表的最末位置, 还没有找到空位置, 则必须从 Hash 表的第一个位置开始再往下找, 如果找到了空位置, 则过程结束; 如果找到刚发生冲突的位置也没有找到空位, 这说明整个 Hash 表都填满了, 产生了存区溢出, 此时, 可另辟一个溢出区把以后输入的记录都存入到这个溢出区中。

在删除时, 不能在 Hash 表中简单地把被删除记录所在位置置空, 否则将影响查找那些

在删除此位置上的记录之前经过此位置而找到空位置进行插入的记录。所以只能对被删除记录所在位置标上已被删的标记，而不是置成空。在以后的插入过程中遇到标上被删标志的位置时，可在此位置上进行插入。

可用下面的过程来描述在 Hash 表中进行各种操作的算法。假设记录的键不超过 10 个字符，用 10 个空格符表示“空”，用 10 个“\*”作为被删标记，但考虑到字符数组结束符，程序中为 9 个“\*”。

---

```
const int b=100; //b 是 Hash 表的桶数
char empty[10]="          ";
char delete[10]="*****";
struct othertype
{ ...; //记录中除键外的数据的类型
};
struct rectype
{ char key[10]; //10 个字符（包括一个结束符）
  othertype other;
};
void makenull(rectype ht[b]) //把 Hash 表 ht 初始化
{ for (int i=0; i < b; i++)
  strcpy(ht[i].key, empty);
}
int search(char x[10], rectype ht[b])
{ //在 Hash 表中查找键为 x 的记录。若找到该记录，则返回值为该记录所在地址
  //否则，返回值为-1
  int i, t;
  t=h(x); //h(x)是所选取的 Hash 函数
  i=0;
  while ((i<b)&&(strcmp(ht[(t+i)% b].key, x)!=0)&&(strcmp(ht[(t+i)%b].key,
    empty)!=0))
    //strcmp 为字符串比较函数，两个字符串相同时返回 0
    i++; t=(t+i)% b;
  if (strcmp(ht[t].key, x)==0) return t; else return -1;
}
void insert(char x[10], othertype y, rectype ht[b])
{ //在 Hash 表 ht 中插入记录(x, y)
  int i, t; t=h(x); i=0;
  while ((i<b)&&(strcmp(ht[(t+i)% b].key, x)!=0)&&(strcmp(ht[(t+i)%b].key,
    empty)!=0)&&(strcmp(ht[(t+i)% b].key, delete)!=0))
    i++; t=(t+i)% b;
  if ((strcmp(ht[t].key, empty)==0) || (strcmp(ht[t].key, delete)==0))
  { ht[t].key=x; ht[t].other=y; }
  else if (strcmp(ht[t].key, x)!=0) cout<<"哈希表满。"<<endl;
}
```

```

void delete(char x[10], rectype ht[b])    //在 Hash 表 ht 中删除键为 x 的记录
{
    int i, t; t=h(x); i=0;
    while ((i< b)&&(strcmp(ht[(t+i)% b].key, x)!=0)&&(strcmp(ht[(t+i)% b].key,
        empty)!=0))
        i++; t=(t+i)% b;
    if(strcmp(ht[t].key, x)==0) strcpy(ht[t].key, delete);
}

```

---

**【例 8-5】** 试编写算法用链地址法解决冲突。

链地址法为另一种解决冲突的方法。当一个冲突发生后，把原桶内无法容纳的记录存放在别的地方，然后用指针把它们链接起来以便检索，在一桶中多次发生冲突后，就形成了一个链表。在具体实现时，Hash 表的各个元素往往都不存放记录，而是存放各个链表的头指针。对于用开式寻址法解决冲突中所使用的键值序列和 Hash 函数，如果采用链地址法解决冲突，就得到如图 8.13 所示的 Hash 表。可利用下面的过程来描述用链地址法解决冲突时在 Hash 表中进行各种操作的算法。

---

```

const int b=26; //b 是 Hash 表中的桶数
const int m=10; //m 为键值的字符个数
struct othertype
{   ...; //othertype 是记录中除键外的数据的类型
};

struct nodetype
{   char key[m];
    othertype other;
    nodetype *link;
};

void makenull(node *ht[b])    //把 Hash 表 ht 初始化
{   for (int i=0; i < b; i++) ht[i]=NULL;   }

nodetype *search(char x[m], nodetype *ht[b])
{//在 Hash 表 ht 中查找键值为 x 的记录。若找到该记录，则返回的值为该记录所在地址
//否则，返回值为 NULL
    nodetype *t; bool found; t=ht[h(x)]; found=false;
    while ((t !=NULL)&&(! found))
        if (strcmp(t->key, x)==0) found=true; else t=t->link;
    return t;
}

void insert(char x[m], nodetype *ht[b], othertype y)
{   //在 Hash 表 ht 中，插入记录(x, y)
    int i; nodetype *t; bool found;
    i=h(x); t=ht[i]; found=false;
    while ((t !=NULL)&&(! found))
        if (strcmp(t->key, x)==0) found=true; else t=t->link;
}

```

```

    if (t==NULL)
    { t=new nodetype; strcpy(t->key, x); t->other=y; t->link=ht[i]; ht[i]=t; }
}

void delete(char x[m], nodetype *ht[b])
{ //删除 Hash 表 ht 中键值为 x 的记录
  int i; nodetype *p, *t; bool found;
  i=h(x); t=ht[i];
  if (t !=NULL)
  { if (strcmp(t->key, x)==0); //x 是第 i 桶中第一个记录的键值
    { p=t; ht[i]=t->tink; delete p; }
    else
    { found=false;
      while ((t->link !=NULL)&&(! found))
      { p=t; t=t->tink;
        if (strcmp(t->key, x)== 0)
        { found=true; p->tink=t->tink; delete t; }
      }
    }
  }
}
}

```

如果已知装填系数为 $\alpha$ ，则可证明：① 若使用开放寻址法解决冲突，则为找到一个结点所需的平均探查次数是  $1/2(1+(1/(1-\alpha)))$ ；② 若使用链地址法解决冲突，则为找到一个结点所需的平均探查次数是  $1+\alpha/2$ 。

## 8.4.4 哈希表的查找及其分析

### 1. 哈希表的查找过程

哈希表的查找过程与其构造过程基本一致，即对于给定的关键字值  $k$ ，根据构造表时设定的哈希函数求得哈希地址，若表中此位置上没有记录，则查找不成功；否则，比较关键字，若和给定值相等，则查找成功；如果不相等，则按照造表时设定的处理冲突的方法求得同义词的“下一地址”，直至求得的哈希地址所指位置为“空”或者表中所填记录的关键字等于给定值  $k$  为止。如果求得的哈希地址所指位置为“空”，则查找失败；如果求得的哈希地址所指位置中存放的关键字的值等于给定值  $k$ ，则查找成功。因此，在查找过程中，影响与给定值进行比较的关键字个数的因素包括哈希函数、冲突处理方法和哈希表的装填因子三个方面。在处理方法相同的哈希表中，平均查找时间主要依赖于哈希表的装填因子。下面给出装填因子的定义：

$$\alpha = \frac{\text{表中填入的记录数}}{\text{哈希表的长度}}$$

其中， $\alpha$  表示哈希表的装满程度。由上式可知， $\alpha$  越小，发生冲突的可能性就越小；反之，说明表中已填入的记录很多，再填记录时发生冲突的可能性就越大。因此在查找过程中，需与给定值进行比较的关键字的个数也就越多。

## 2. 哈希表查找过程的分析——平均查找长度

从哈希表的查找过程可看出,利用关键字进行哈希地址计算后,由于存在冲突,哈希表的查找过程即为与关键字的比较过程,因此可利用平均查找长度来衡量哈希表的查找效率。下面给出不同的探测再散列方法在各种情况下的平均查找长度(推导过程省略)。

(1) 查找成功时的平均查找长度

① 线性探测再散列 
$$S_{nl} \approx \frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right)$$

② 伪随机探测再散列、二次探测再散列和再哈希表 
$$S_{nr} \approx \frac{1}{\alpha} \ln(1-\alpha)$$

③ 链地址法 
$$S_{nc} \approx 1 + \frac{\alpha}{2}$$

(2) 查找不成功时的平均查找长度

① 线性探测再散列 
$$U_{nl} \approx \frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right)$$

② 伪随机探测再散列、二次探测再散列和再哈希表 
$$U_{nr} \approx \frac{1}{1-\alpha}$$

③ 链地址法 
$$U_{nc} \approx \alpha + e^{-\alpha}$$

由以上结果可知,哈希表的平均查找长度是 $\alpha$ 的函数,而不是 $n$ 的函数,因此不管 $n$ 为多大,总可选择一个合适的装填因子以便将平均查找长度限定在一个范围内。

# 本章总结

## 1. 基本内容

本章主要介绍了静态表的4种查找方法(顺序表查找、有序表的折半查找、静态树表的折半查找、索引顺序表查找),动态树的4种查找方法(二叉排序树、二叉平衡树、B-树、B<sup>+</sup>树),哈希表和哈希函数的构造方法,并进行了衡量各种查找效率的平均查找长度的讨论,主要学习要点如下:

- ① 查找的定义、分类和各类查找的特点;
- ② 顺序查找和折半查找的思想和实现方法;
- ③ 二叉排序树和平衡二叉树的概念和有关运算的实现方法;
- ④ 哈希表(散列表)的基本思想、哈希函数(散列出数)构造及处理冲突的方法和算法;
- ⑤ 各类查找算法的时间复杂度分析和平均查找长度的计算方法。

## 2. 基本要求

(1) 深刻领会查找的基本概念及静态查找的实现

- ① 弄清集合、元素及属于关系等基本概念及作为逻辑结构的集合概念;
- ② 知道静态和动态查找表的定义、区别、分类及应用背景;
- ③ 清楚查找表的逻辑结构是集合以及它的基本特点;
- ④ 掌握静态查找表的顺序表示方法和查找算法;

⑤ 掌握静态找表的折半和分块查找算法、限制条件、算法时间复杂度。

(2) 掌握动态查找表的分类及实现方法

① 掌握二叉排序树的定义、性质、查找算法及插入运算的实现方法;

② 掌握平衡二叉树的定义、生成过程、平衡旋转的 4 种类型及实现;

③ 知道 B-树, B<sup>+</sup>树的区别及其查找实现方法和分析。

(3) 深刻理解哈希表的基本概念和其存储组织及处理冲突的方法

① 清楚哈希函数、哈希表的概念及其特点;

② 掌握哈希函数的构造方法和解决冲突的基本方法;

③ 知道开哈希函数和闭哈希函数的相对优缺点;

④ 掌握开哈希函数的存储组织方法及处理冲突方法;

⑤ 掌握在开哈希表上实现查找、插入、删除运算的基本思想和算法;

⑥ 掌握在闭哈希表上实现查找和插入的基本思想和算法。

### 3. 重点与难点

本章重点为: 折半、二叉排序树、哈希表等的查找算法。难点为: 索引顺序查找及哈希函数构造及处理冲突的方法。

## 习题 8

8-1 设一个有序文件的关键字为 2, 5, 8, 11, 13, 17, 25, 36, 47, 55, 60, 63, 65, 67, 76, 84, 当用折半查找算法查找关键字 2, 55, 63 时, 试决定关键字的比较次数。

8-2 在折半查找算法中, 如果做下述之一的变动, 它是否还能正确地工作:

(1) 将语句  $\text{mid} = (\text{low} + \text{high}) \text{ DIV } 2$  改为  $\text{mid} = \text{low}$ ;

(2)  $\text{low} = \text{mid} + 1$  改成  $\text{low} = \text{mid}$ ;

(3) 将  $\text{high} = \text{mid} - 1$  改成  $\text{high} = \text{mid}$ ;

(4) 同时做 (2) 和 (3) 中的变动。

8-3 用 C++ 语言改写折半查找算法, 使其采用递归的调用。

8-4 假定  $x_1 < x_2 < \dots < x_n$  形成一棵二叉查找树, 对某个  $i \geq 2$  和  $x_i$ , 若有结点  $x_i$  的左子树为空, 证明: 当用二叉排序树查找的算法来查找  $x_i$  时, 必将对  $x_{i-1}$  进行一次探查。

8-5 假设  $x_1 < x_2 < \dots < x_n$  形成一棵二叉查找树, 证明: 当用二叉排序树查找算法查找  $z$ , 且  $x_i < z < x_{i+1}$  时, 必将对  $x_i$  和  $x_{i+1}$  进行探查。

8-6 假设一棵平衡二叉树的每个结点都标明了平衡因子, 试设计一个算法, 求平衡二叉树的高度。

8-7 在一棵有  $n$  个结点的平衡二叉树中, 至少有多少个平衡因子为零的结点。

8-8 (1) 写出二叉搜索树的插入算法, 往二叉搜索树中插入一个新元素  $k$ ; (2) 写出从二叉查找树中删除一个结点的算法。要求用它的前驱结点取代它在原树中的位置而不是后继结点。

8-9 假定二叉搜索树的结点定义如下:

---

```
struct  nodetype { int k, r;
                  nodetype *llink, *rlink;
};
```

---



其中,  $k$  和  $r$  分别是结点的键和信息,  $llink$  和  $rlink$  是它的左、右孩子。写出一个 C++ 函数  $Search(tree, key, rec)$ , 它查找以  $tree$  为根结点的二叉查找树的具有键值为  $key$  的记录  $rec$ 。

8-10 若 Hash 向量长度为 1000, 如果 Hash 函数只是简单地抽取关键字的内部代码中间三位数字作为 Hash 函数值, 会出现什么问题?

8-11 试比较开放地址法和链地址法冲突处理技术的优缺点。

8-12 若采用链地址法解决冲突, 试写出从 Hash 表中删除某一记录  $k$  的算法。

8-13 假设结点序列  $F = (60, 30, 90, 50, 120, 70, 40, 80)$ 。试用查找树的插入算法, 用  $F$  中的结点依次进行插入, 画出  $F$  中结点所构成的查找树  $t_1$ ; 再用查找树的删除算法, 从查找树  $t_1$  中依次删除 40, 70, 60, 画出删除后的查找树  $t_2$ 。

8-14 假设  $T$  是按标准形式存储的二叉树, 树中的结点值为序数, 试编写一个判断二叉树  $T$  是否为查找树的程序过程。

8-15 试编写一个判断给定二叉树  $T$  是否为平衡树的程序过程。

8-16 试用 Adelson 插入方法依次把结点 50, 20, 10, 100, 120, 30, 110, 60, 70, 90, 80, 40 插入到初始时为空的平衡查找树中, 使得在每次插入后保持该树仍然是平衡查找树。请依次画出每次插入后所形成的平衡查找树。

8-17 (1) 编写一个在 B\_树中插入结点的程序过程; (2) 编写一个在 B\_树中删除结点的程序过程。

8-18 某校 84 级同学举办运动会, 报名参加的同学为 84438, 84102, 84528, 84136, 84338, 84250, 84407, 84239, 84227, 84517, 84321, 84421, 84451, 84241, 84118, 84543, 84309, 画出进行分块查找的数据组织形式。

8-19 试编写一个开放定址法解决冲突的哈希表删除方法。

8-20 设有 10 个记录的关键字为 ICKES, BARBER, ELYOT, KERN, FRENCH, LOWES, BENSDN, FONK, ERVIN, KNOX, 试构造  $\alpha = 10/13$  的哈希表, 取关键字首字母在字母表中的序号为哈希函数值, 随机探测解决冲突,  $d_j = (d_1 + R_j) \text{MOD } 13$ ,  $R_j$  取自伪随机序列: 3, 17, 1, 12, 10, ..., 统计该表的平均查找长度 ASL。

8-21 设有关键字分别为 A、B、C、D 的 4 个元素, 按照不同的输入顺序, 画出所有可能的二叉查找树的图形。

8-22 假定用二叉查找树表示数据集合, 试写出实现操作  $search(k, f)$ ,  $insert(k, f)$  的非递归程序。其中,  $k$ ,  $r$  分别表示已知的关键字和记录,  $f$  表示二叉查找树根结点的指针。

8-23 设计一个结构, 存放学生的竞赛结果, 然后对所有学生输出各人的名次, 每个学生的信息应包含以下数据项: 学生姓名、文法成绩、笔译成绩、口语成绩、总分。(1) 要求以下列两种方式输出: (a) 按名次顺序将所有学生信息输出; (b) 按姓名的汉语拼音字母顺序输出所有学生信息, 并列出具体的名次。(2) 用折半检索法, 检索指定学生的成绩, 总分及名次, 若找不到, 应显示相关信息。

8-24 设线性表中共有  $n$  个结点, 而第  $i$  个结点的使用频率为  $p_i = 1/2^i$  ( $1 \leq i \leq n$ ), 试求出按顺序查找法在线性表中进行一次成功查找所需的平均比较次数。

8-25 假设线性表中结点的使用频率预先并不知道, 为了使得经常被查找的结点尽量放在线性表的前面, 可采用如下的策略: 若找到指定的结点, 则把此结点向表头方向前移一个位置。试对顺序分配的线性表和链接分配的线性表分别写出实现上述策略的顺序查找法的程序过程。

8-26 试编写一个寻找指定结点在给定的查找树中所在的层次的程序过程。

8-27 作为输入的是  $n$  ( $n \leq 1000$ ) 英文单词, 要为这  $n$  个单词建立一个 Hash 表。所取的 Hash 函数为  $h(\text{first}(x)) = \text{ord}(\text{first}(x)) - \text{ord}('a')$ 。其中  $x$  为某个输入单词,  $\text{first}(x)$  取  $x$  所代表的单词的第一个英文字母, 这样的 Hash 表共有 26 个桶号 (0, 1, ..., 25)。要建立的 Hash 表结构与链地址法解决冲突的结构相似, 所不同的是用查找树代替链表。请按上述条件和要求编写一个程序过程, 用它建立所需的 Hash 表。

# 第9章 内部排序

排序是数据处理过程中常用的一种重要运算，如何进行高效率的排序是计算机软件理论研究的重要课题之一。排序方法按照排序过程中所涉及的存储器可分为内排序和外排序两种，其中，待排序记录全部存放在计算机内存中进行排序的过程，称为内排序；而由于待排序记录的数量很大使得排序过程中也需要对外存设备进行访问的排序过程，称为外排序，本章主要讨论内排序的几种主要算法和它们的相关性能。

## 9.1 排序的基本概念

### 1. 相关术语和概念

#### (1) 排序

所谓排序就是整理文件中的记录次序，使其按关键码递增（或递减）次序进行排列，其形式化定义如下：

输入： $n$  个记录  $R_1, R_2, \dots, R_n$ ，其相应的关键码分别为  $K_1, K_2, \dots, K_n$ 。

输出： $R'_1, R'_2, \dots, R'_n$ ，使得  $K'_1 \leq K'_2 \leq \dots \leq K'_n$ （或  $K'_1 \geq K'_2 \geq \dots \geq K'_n$ ）。

即  $R'_1, R'_2, \dots, R'_n$  为  $R_1, R_2, \dots, R_n$  的一种排列，该排列按照其相应的关键码满足  $K'_1 \leq K'_2 \leq \dots \leq K'_n$  的条件生成。

#### (2) 排序算法的稳定性

在待排序文件中，若存在多个关键码相同的记录，经过排序后，这些具有相同关键码的记录之间的相对次序保持不变，则称该排序方法是稳定的；否则，称这种排序方法是不稳定的。

#### (3) 排序算法的两种基本操作

①比较两个关键码的大小；②将记录从一个位置移动至另一个位置。

### 2. 数据类型

为了方便讨论，待排记录的数据类型定义如程序 9-1 所示，读者可自行将其设计为类。

程序 9-1 待排序记录的数据类型定义

```
#define MAXSIZE 20 //一个用作示例的小顺序表的最大长度
template<class T>
struct RedType{
    KeyType key; //关键码项
    T otherinfo; //其他数据项
};
template<class T> //记录类型
struct Sqlist{
```

```
RedType<T> r[MAXSIZE+1]; //r[0]闲置或用作哨兵单元
int length; //顺序表长度
}; //顺序表类型
```

---

### 3. 排序算法的效率

评价排序算法的效率主要有两点。①在数据规模一定的条件下，算法执行所消耗的平均时间。对于排序操作，时间主要消耗在关键码之间的比较和数据元素的移动上，因此可认为高效率的排序算法应该有尽可能少的比较次数和尽可能少的数据元素移动次数。②在数据规模一定的条件下，执行算法所需要的辅助存储空间。辅助存储空间是除了存放待排序数据元素占用的存储空间之外，执行算法所需要的其他存储空间。理想的空间效率是，算法执行期间所需要的辅助空间与待排序的数据量无关。

## 9.2 插入排序

插入排序（insertion sort）的基本思想是，将待排序的记录，按其关键码的大小插入到已经排好序的有序子表中，直到全部记录插入完成为止。通常需要一个辅助空间来存储当前待插入的记录，在插入时，需要移动已排序记录，为新插入的元素提供空间。下面主要介绍直接插入排序、折半插入排序、2路插入排序、表插入排序及希尔排序。

### 9.2.1 直接插入排序

#### 1. 算法思想

直接插入排序（straight insertion sort）是一种比较简单的排序方法，它的基本思想是，依次将记录序列中的每一个记录插入到有序段中，使有序段的长度不断地扩大。算法实现思路是，先将待排序记录序列中的第一个记录作为一个有序段，将记录序列中的第二个记录插入到上述有序段中形成的由两个记录组成的有序段，再将记录序列中的第三个记录插入到这个有序段中，形成由三个记录组成的有序段，……，其余类推，直到所有记录都插入到有序段中为止。一共需要经过  $n-1$  趟就可将初始序列的  $n$  个记录重新排列成按关键码值大小排列的有序序列。具体实现过程见程序 9-2，其中，监视哨为  $L.r[0]$ ，它的作用是：保存每趟排序过程中的  $L.r[i]$  的副本，防止在记录后移时丢失  $L.r[i]$  的内容；在第二个 `do_while` 循环中“监视”下标变量  $j$  是否越界，一旦越界（即  $j=0$ ），则循环判定条件不成立，结束循环。

#### 2. 算法实现

程序 9-2 直接插入排序算法

---

```
template<class T>
void InsertSort(SqList<T> &L) {
    //对顺序表 L 进行直接插入排序
    int i, j;
    for (i=2; i<=L.length; i++)
        if (L.r[i].key<L.r[i-1].key)
```

```

{   L.r[0]=L.r[i]; j=i-1; //复制为哨兵
    do { L.r[j+1]=L.r[j]; j--; //记录后移
        }while (j>=1 && L.r[0].key<L.r[j].key);
    L.r[j+1]=L.r[0]; //插入到正确位置
}
}; //InsertSort

```

**【例 9-1】** 已知待排序的一组记录的关键码初始排列：52, 43, 68, 89, 77, 16, 24, 52\*, 80, 31。按程序 9-2 进行直接插入排序的过程如图 9.1 所示。

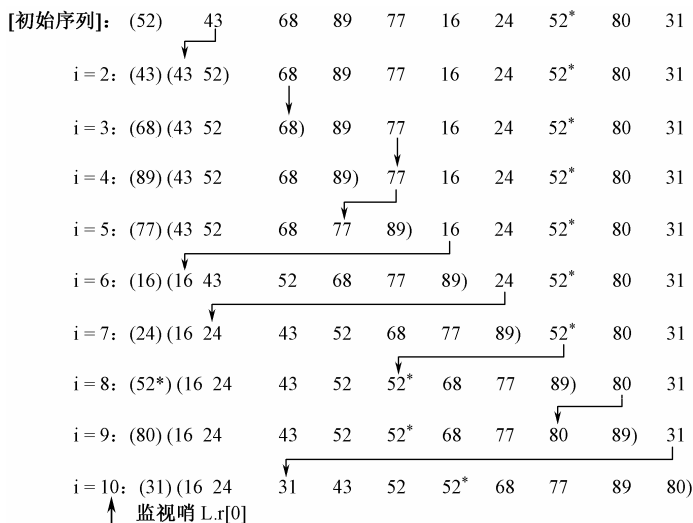


图 9.1 直接插入排序示例

如图 9.1 所示的过程可看出，对于相同的两个关键码 52 和 52\*，直接插入排序后并没有改变这两个关键码原来的相对次序，因此该排序算法是稳定的。

### 3. 算法效率

直接插入排序算法比较简单，从空间效率来看，仅用到一个记录的辅助空间作为监视哨。从时间效率来看，排序的基本操作主要为比较两个关键码的大小和移动记录，可分以下三种情况进行分析。

① 最好情况：当待排序列中记录按关键码非递减有序排序（“正序”）时，所需进行关键码间比较的次数达最小值  $n-1$ （即  $\sum_{i=2}^n 1$ ），即记录不需移动。

② 最坏情况：当待排序列中记录按关键码非递增有序排序（“逆序”）时，总的比较次数达最大值  $(n+2)(n-1)/2$ （即  $\sum_{i=2}^n i$ ），记录移动的次数也达最大值  $(n+4)(n-1)/2$ （即

$\sum_{i=2}^n (i+1)$ ）。

③ 平均情况：若待排序记录是随机的，即待排序列中的记录可能出现的各种排列概率

相同，则取上述最小值和最大值的平均值，作为直接插入排序时所需进行关键码间的比较次数和移动记录的次数，约为  $n^2/4$ 。

综上所述，直接插入排序的时间复杂度为  $O(n^2)$ 。

### 9.2.2 折半插入排序

由于插入排序的基本思想是将待插入的新记录插入到已排序的有序表中，因此可在查找插入位置时采用折半查找的方法。折半插入排序（binary insertion sort）的查找过程就是利用“折半查找”来实现的，算法实现过程见程序 9-3。

程序 9-3 折半插入排序算法

```
template<class T>
void BInsertSort(SqList<T> &L) {
//对顺序表 L 作折半插入排序
    int i, j, low, high, middle;
    for (i=2; i<=L.length; ++ i) { L.r[0]=L.r[i]; //将 L.r[i] 暂存到 L.r[0] 中
        low=1; high=i-1;
        while (low <=high) { //在 r[low..high] 中折半查找有序插入的位置
            middle=(low + high)/ 2; //折半
            if (L.r[0].key<L.r[middle].key) high=middle-1; //插入点在低半区
            else low=middle + 1; //插入点在高半区
        } //while
        for (j=i-1; j >=high + 1; --j) L.r[j + 1]=L.r[j]; //记录后移
        L.r[high + 1]=L.r[0]; //插入
    } //for
} //BInsertSort
```

时间复杂度为  $O(n^2)$ 。

### 9.2.3 2 路插入排序

2 路插入排序是在折半插入排序的基础上进行改进的，其目的是减少排序过程中移动记录的次数。算法实现的思路是，设一个和 L.r 同类型的数组 d，首先将 L.r[1] 赋值给 d[1]，并将 d[1] 看成在排好序的序列中处于中间位置的记录，然后从 L.r 中第 2 个记录起依次插入到 d[1] 之前或之后的有序序列中。先将待插入记录的关键码和 d[1] 的关键码进行比较，若  $L.r[i].key < d[1].key$ ，则将 L.r[i] 插入到 d[1] 之前的有序表中；反之，则将 L.r[i] 插入到 d[1] 之后的有序表中。实现算法时，可将 d 看成是一个循环向量，并设两个指针 first 和 final 分别指示排序过程中得到的有序序列中的第一个记录和最后一个记录在 d 中的位置。该算法移动记录的次数约为  $n^2/8$ 。

**【例 9-2】** 以关键码序列 {52, 43, 68, 89, 77, 16, 24, 52\*, 80, 31} 为例，进行 2 路插入排序的过程如图 9.2 所示。

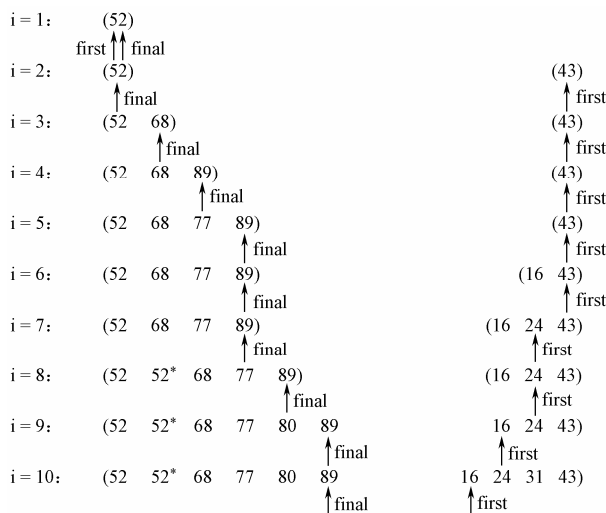


图 9.2 2 路插入排序示例

## 9.2.4 表插入排序

虽然上述 2 路插入法的比较次数比直接插入排序的要少，但移动次数相等，若希望在排序过程中不移动记录，则只有改变存储结构才能实现。下面介绍的表插入算法便是在不移动记录的情况下利用存储结构有关信息的改变来实现排序的，为此需要为每个记录设置一个类型为整型的指针域。排序算法的基本思路是，在插入第  $i$  个记录  $R_i$  时， $R_1, R_2, \dots, R_{i-1}$  已经通过各自的指针域按关键码的值递增的次序连接成一个静态表，将  $R_i$  的关键码的值与表中已排好序的关键码值从表头开始向后依次进行比较，找到  $R_i$  的插入位置后插入，此时表中各记录的关键码值仍然保持有序，程序 9-4 给出所用的存储结构定义。

程序 9-4 表插入排序的记录存储结构定义

```
#define SIZE 100 //静态链表容量

template<class T>
struct SLNode{
    RedType<T> rc; //记录项
    int next; //指针项，指示下一元素的位置
}; //表结点类型

template<class T>
struct SLinkListType{
    SLNode<T> r[SIZE]; //0 号单元为表头结点
    int length; //链表当前长度
}; //静态链表类型
```

以上述静态链表类型作为待排记录序列的存储结构，并设数组中下标为 0 的分量为表头结点，并令表头结点记录的关键码取最大整数 MAXNUM。

表插入的排序过程是，首先将静态链表中数组下标为 1 的分量（结点）和表头结点构成一个循环链表，然后依次将下标为 2~ $n$  的分量（结点）按记录关键码非递减有序插入到循环链表中，算法如程序 9-5 所示。

```
#define MAXNUM10000
template <class T>
void TableInsertSort(SLinkListType<T> &SL)
{   int head=0, pre, cur, i;
    SL.r[0].rc.key=MAXNUM10000;  //置表头关键码为最大值
    SL.r[0].next=0;  //置表头指针
    for (i=1; i<=SL.length; i++) {
        if (SL.r[head].rc.key>SL.r[i].rc.key) {
            SL.r[i].next=head; head=i; SL.r[0].next=head;  //更改表头指针
        } //if
        else {   for (cur=head; cur!=0 && SL.r[cur].rc.key<=SL.r[i].rc.key;
                    cur=SL.r[cur].next) pre=cur; SL.r[pre].next=i; SL.r[i].next=cur;
                } //else
    } //for
}
```

【例 9-3】 以关键码序列 {52, 43, 68, 89, 77, 16, 24, 52\*, 80, 31} 为例，表插入排序的过程如图 9.3 所示。

初始状态	Key域	MAXNUM	52	43	68	89	77	16	24	52*	80	31
	Link域	1	0	—	—	—	—	—	—	—	—	—
i=2	Key域	MAXNUM	52	43	68	89	77	16	24	52*	80	31
	Link域	2	0	1	—	—	—	—	—	—	—	—
i=3	Key域	MAXNUM	52	43	68	89	77	16	24	52*	80	31
	Link域	2	3	1	0	—	—	—	—	—	—	—
i=4	Key域	MAXNUM	52	43	68	89	77	16	24	52*	80	31
	Link域	2	3	1	4	0	—	—	—	—	—	—
i=5	Key域	MAXNUM	52	43	68	89	77	16	24	52*	80	31
	Link域	2	3	1	5	0	4	—	—	—	—	—
i=6	Key域	MAXNUM	52	43	68	89	77	16	24	52*	80	31
	Link域	6	3	1	5	0	4	2	—	—	—	—
i=7	Key域	MAXNUM	52	43	68	89	77	16	24	52*	80	31
	Link域	6	3	1	5	0	4	7	2	—	—	—
i=8	Key域	MAXNUM	52	43	68	89	77	16	24	52*	80	31
	Link域	6	8	1	5	0	4	7	2	3	—	—
i=9	Key域	MAXNUM	52	43	68	89	77	16	24	52*	80	31
	Link域	6	8	1	5	0	9	7	2	3	4	—
i=10	Key域	MAXNUM	52	43	68	89	77	16	24	52*	80	31
	Link域	6	8	1	5	0	9	7	10	3	4	2

图 9.3 表插入排序过程

从上面的过程可看出，表插入排序与直接插入排序的不同之处在于，表插入排序以修

改  $2n$  次指针值来代替记录的移动，但和关键码的比较次数相同，因此表插入排序的时间复杂度仍为  $O(n^2)$ 。

### 9.2.5 希尔排序

希尔排序 (shell sort) 又称“缩小增量排序”，是插入排序的一种，因 Shell 于 1959 年提出而得名。算法思想是，将待排序的记录划分成几组，从而减少参与直接插入排序的数据量，当经过几次分组排序后，记录的排列已经基本有序，这个时候再对所有的记录实施直接插入排序。具体步骤描述如下：假设待排序的记录为  $n$  个，先取整数  $d < n$ ，例如，取  $d = \lfloor n/2 \rfloor$ ，将所有距离为  $d$  的记录构成一组，从而整个待排序记录序列被分割为  $d$  个子序列，对每个分组分别进行直接插入排序；然后再缩小间隔  $d$ ，例如，取  $d = \lfloor d/2 \rfloor$ ；重复上述过程，再对每个分组分别进行直接插入排序，直到最后取  $d = 1$ ，即将所有记录放在一组进行一次直接插入排序，最终将所有记录重新排列成按关键码有序的序列。程序 9-6 为希尔排序算法。

程序 9-6 希尔排序算法

```
template <class T>
void ShellSort(SqList<T> &L, int L.length)
{
    int i, j, dk=L.length; //增量的初值
    do { dk=dk/2; //求下一个增量
        for (i=dk+1; i<=L.length; i++) //各子序列交替处理
            if (L.r[i]<L.r[i-dk]) { //逆序
                L.r[0]=L.r[i]; j=i-dk;
                do { L.r[j+dk]=L.r[j]; //后移元素
                    j=j-dk; //再比较前一元素
                }while (j>=1 && L.r[0]<L.r[j]);
                L.r[j+dk]=L.r[0]; //将 L.r[i] 回送
            }
        }while(dk>1);
    };
};
```

**【例 9-4】** 以关键码序列 {52, 43, 68, 89, 77, 16, 24, 52\*, 80, 31} 为例，说明希尔排序的过程。初始关键码序列如图 9.4 的第 1 行所示。首先，将该序列分成 5 个子序列：{ $R_1, R_6$ }, { $R_2, R_7$ }, ..., { $R_5, R_{10}$ }，分别对每个子序列进行直接插入排序。然后，将第 1 趟希尔排序的结果分成 3 个子序列：{ $R_1, R_4, R_7, R_{10}$ }, { $R_2, R_5, R_8$ } 和 { $R_3, R_6, R_9$ }，并对它们进行直接插入排序。最后，对整个序列进行一趟直接插入排序。至此，希尔排序结束，整个序列的记录已按关键码非递减有序排列。

由上面的例子可看出，希尔排序是一个不稳定的排序方法，除此以外，由于它是通过不断缩小增量（步长）而将原始序列分成若干个子序列的，这使得序列变得越来越有序，从而提高插入排序的效率。希尔排序中的步长有各种不同的取法，一般认为，步长都取成奇数且步长之间互素比较好。但如何选取步长为最好，至今在理论上仍没有得到论证，只是需要注意，步长因子中除 1 外没有公因子，且最后一个步长因子必须是 1。



[初始关键码]: 52 43 68 89 77 16 24 52\* 80 31

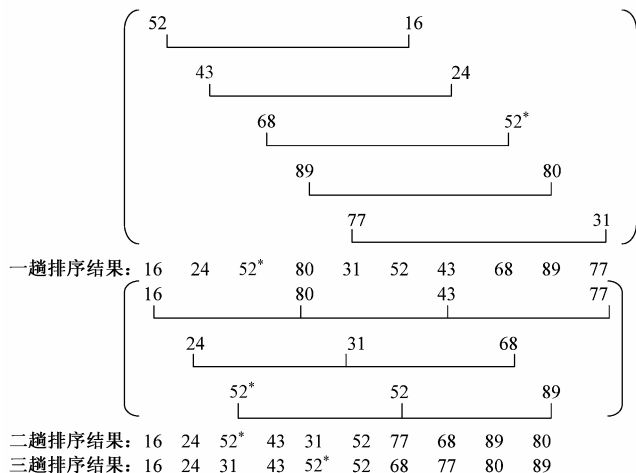


图 9.4 希尔排序示例

## 9.3 交 换 排 序

交换排序的基本思想是，将待排序记录的关键码进行两两比较，发现两个记录的次序相反时即进行交换，直到没有反序的记录为止。下面主要介绍冒泡排序和快速排序两种方法。

### 9.3.1 冒泡排序

冒泡排序（bubble sort）是交换排序中一种简单的排序方法，其基本思想是对所有相邻记录的关键码值进行比较，如果是逆序（ $L.r[1].key > L.r[2].key$ ），则将其交换，最终达到有序化。其处理过程为：①将整个待排序的记录序列划分成有序区和无序区，初始状态时，有序区为空，无序区包括所有待排序的记录；②对无序区从前向后依次将相邻记录的关键码进行比较，若逆序，则将其交换，从而使得关键码值小的记录向上“飘浮”（左移），关键码值大的记录向下“坠落”（右移）。

每经过一趟冒泡排序，都使无序区中关键码值最大的记录进入有序区，对于由  $n$  个记录组成的记录序列，最多经过  $n-1$  趟冒泡排序，就可将这  $n$  个记录重新按关键码顺序排列。可看出，若“在一趟排序过程中没有进行过交换记录的操作”，则可结束整个排序过程。

**【例 9-5】** 如图 9.5 所示为冒泡排序的一个示例。

冒泡排序的效率分析：① 若初始序列为“正序”序列，则只需进行一趟排序，在排序过程中进行  $n-1$  次关键码的比较，且不移动记录；② 若初始序列为“逆序”序列，则需要进行  $n-1$  趟排序，需进行  $\sum_{i=n}^2 (i-1) = n(n-1)/2$  次比较。因此，总的时间复杂度为  $O(n^2)$ 。

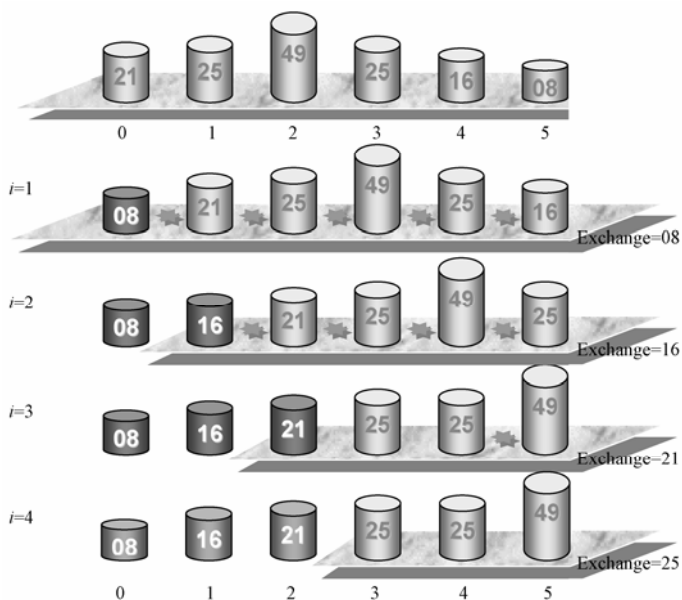


图 9.5 冒泡排序示例

### 9.3.2 快速排序

快速排序 (quick sort) 是 Hoare 于 1962 年提出的一种分区交换排序。它采用了一种分而治之的策略，是对冒泡排序的一种改进。其算法思想是，首先将待排序记录序列中的所有记录作为当前待排序区域，选取第一个记录的关键码值为基准 (轴值)，从待排序记录序列左、右两端开始向中间靠拢，交替与轴值进行比较，若左侧记录的关键码值大于轴值，则将该记录移到基准记录的右侧，若右侧记录的关键码值小于轴值，则将该记录移到基准记录的左侧，最后让基准记录到达它的最终位置，此时，基准记录将待排序记录分成了左右两个区域，位于基准记录左侧的记录都小于或等于轴值，位于基准记录右侧的所有记录的关键码都大于或等于轴值，这就是一趟快速排序；然后分别对左右两个新的待排序区域，重复上述一趟快速排序的过程，其结果是分别让左、右两个区域中的基准记录都到达它们的最终位置，同时将待排序记录序列分成更小的待排序区域，再次重复对每个区域进行一趟快速排序，直到每个区域只有一个记录为止，此时所有的记录都到达了它的最终位置，即整个待排序记录按关键值有序排列，至此排序结束。

快速排序的算法参见程序 9-7、程序 9-8 和程序 9-9。

#### 程序 9-7 快速排序算法的描述

```

QSort(SqList &L, int low, int high) {
    if (顺序表 L 的长度大于 1) {
        将顺序表 L 分为两个子表序列;
        分别对两个子表序列进行排序;
        将两个子表序列合并为一个序列 L;
    }
}

```

```

Void QSort(SqlList &L, int low, int high)
{
    //对顺序表 L 中的序列 L.r [low..high] 作快速排序
    if (low < high) { //表 L 长度大于 1
        pivotloc=Partition (L, low, high) //将序列 L.r 再划分为两个子序列
        //分别对两个子序列进行排序
        QSort(L, low, pivotloc-1);
        QSort(L, pivotloc+1, high);
    }
}

```

程序 9-9 一趟快速排序算法

```

Template <class T>
int Partition (SqlList <T> &L, int low, int high) {
    //交换顺序表 L 中 L.r[low..high] 的记录, 使枢轴记录到位, 并返回其所在位置
    //此时在它之前(后)的记录均不大于(小于)它
    int pivotloc=low; L.r[0]=L.r[low]; // L.r[0] 是基准元素(枢轴记录)
    for (int i=low+1; i<=high; i++) //检测整个子序列, 并进行划分
        if (L.r[i] < L.r[low]) {
            pivotloc++;
            if (pivotloc !=i)
                Swap (L.r[pivotloc], L.r[i]); //小于基准元素的值交换到左侧
        }
    L.r[low]=L.r[pivotloc];
    L.r[pivotloc]=L.r[0]; //将基准元素(枢轴记录)就位
    Return pivotloc; //返回枢轴记录位置
}

```

**【例 9-6】** 以关键码序列{21, 25, 49, 25\*, 16, 08}为例, 一趟快速排序的过程如图 9.6 (a) 所示, 整个序列的快速排序过程如图 9.6 (b) 所示。

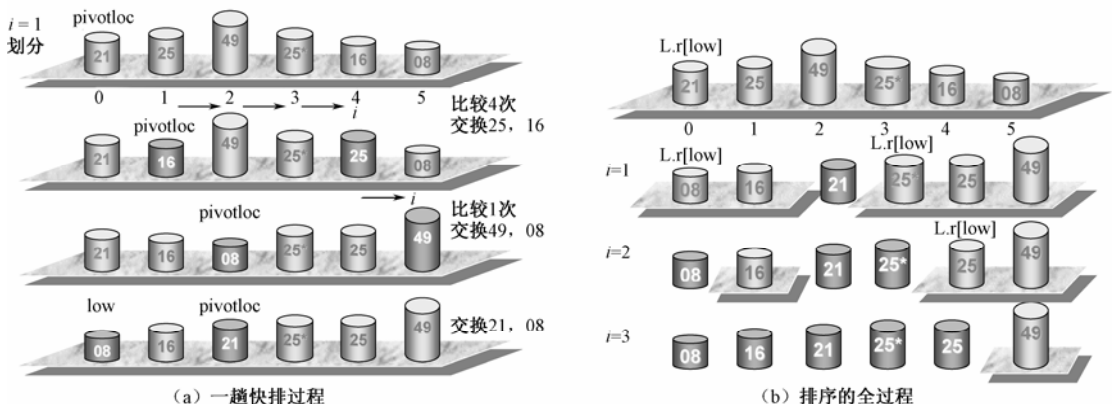


图 9.6 快速排序示例

从本例可看出，快速排序方法是一个不稳定的排序方法。快速排序通常被认为是具有相同数量级  $O(n\log_2 n)$  的排序方法中平均性能最好的方法，但是，如果初始序列中关键码是有序或基本有序的，则快速排序方法与冒泡排序的效率接近，比较低。

## 9.4 选择排序

选择排序（selection sort）的基本思想是，每一趟从待排序的记录中选出关键码最小的记录，顺序放在已排好序的子序列后面，直到全部记录排序完毕。

### 9.4.1 简单选择排序

简单选择排序的算法思想是每一趟在  $n-i+1$  ( $i=1, 2, 3, \dots, n-1$ ) 个记录中选取关键码最小的记录作为有序序列中的第  $i$  个记录。

具体操作步骤如下。

① 将整个记录序列划分为有序区域和无序区域，有序区域位于最左端，无序区域位于右端，初始状态有序区域为空，无序区域含有待排序的所有  $n$  个记录。

② 设置一个整型变量 `index`，用于记录在一趟的比较过程中，当前关键码值最小的记录位置。开始将它设定为当前无序区域的第一个位置，即假设这个位置的关键码最小，然后用它与无序区域中其他记录进行比较，若发现有比它的关键码还小的记录，就将 `index` 改为这个新的最小记录位置，随后再用 `L.r[index].key` 与后面的记录进行比较，并根据比较结果，随时修改 `index` 的值。一趟结束后，`index` 中保留的就是本趟选择的关键码最小的记录位置。

③ 将 `index` 位置的记录交换到无序区域的第一个位置，使得有序区域扩展了一个记录，而无序区域减少了一个记录。

不断重复步骤②、③，直到无序区域剩下一个记录为止，此时所有的记录已经按关键码从小到大的顺序排列就位。实现过程如程序 9-10 所示。

程序 9-10 简单选择排序算法

```
template<class T>
SelectMinKey(SqList<T> &L, int &i)
{//在中 L.r[1..L.length]选择 key 最小的记录
    int min=L.r[i].key; int p=i;
    for (int k=i+1; k<=L.length; k++)
    {   if (L.r[k].key<min) {   min=L.r[k].key; p=k;   }}
    return p;
}

template<class T>
void SelectSort(SqList<T> &L) {
    //对顺序表 L 作简单选择排序
    int j;
    for (int i=1; i<L.length; ++i) { //选择第 i 小的记录，并交换到位
        j=SelectMinKey(L, i); //在 L.r[1..L.length]中选择 key 最小的记录
        RedType<T> temp;
        if (i!=j)
```

```

        {   temp=L.r[i]; L.r[i]=L.r[j]; L.r[j]=temp; } //与第 i 个记录交换
    } //for
} //SelectSort

```

简单选择排序的比较次数与待排序序列的初始状态无关，在第  $i$  趟排序过程中需要进行  $n-i$  次比较，因此总的比较次数为  $O(n^2)$ ，总的移动次数为  $n-1$ ，因此它的平均时间复杂度为  $O(n^2)$ 。同时，简单选择排序也是不稳定的排序算法。

## 9.4.2 堆排序

### 1. 定义

下面先给出相关定义。

(1) 堆是指  $n$  个元素的序列  $\{k_1, k_2, \dots, k_n\}$  当且仅当满足以下关系时，称为堆，其中满足关系①的称为最小堆，满足关系②的称为最大堆。

$$\textcircled{1} \begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \quad \text{或} \quad \textcircled{2} \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases} \quad (i=1, 2, \dots, \lfloor n/2 \rfloor)$$

例如，下列两个序列为堆  $\{87, 46, 75, 23, 14, 39, 52, 08, 20\}$ ， $\{14, 26, 37, 32, 58, 71\}$ ，对应的完全二叉树如图 9.7 所示。

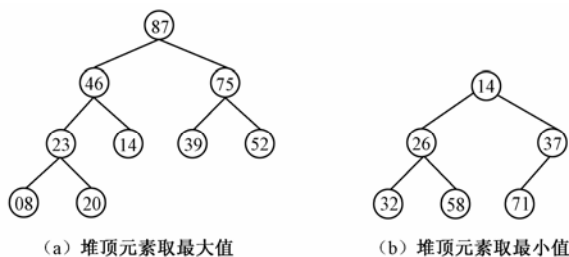


图 9.7 堆的示例

(2) 堆排序 (heap sort) 是指若在输出堆顶的最大 (小) 值之后，使得剩余  $n-1$  个元素的序列重又建成一个堆，则得到  $n$  个元素中的次大 (小) 值。如此反复执行，便能得到一个有序序列，这个过程称为堆排序。堆排序只需要一个记录大小的辅助空间，每个待排序的记录仅占有一个存储空间。

### 2. 最大堆与它的主要操作

程序 9-11 给出了最大堆的类定义。其中， $n$  是私有成员，代表目前堆中元素的个数；MaxSize 是堆的最大容量；heap 为存储堆元素的数组，默认堆的大小为 10 个元素。最小堆的类实现与最大堆的类似，作为练习，请读者自行完成。

程序 9-11 类 MaxHeap

```

template<class T>
class MaxHeap {
public:
    MaxHeap(int MaxHeapSize=10);
    ~MaxHeap() { if (heap) delete [ ] heap; }

```

```

int Size() const { return CurrentSize; }
T Max() { if (CurrentSize==0) throw OutOfBounds();
return heap[1]; }
MaxHeap<T>& Insert(const T& x); MaxHeap<T>& DeleteMax(T& x);
void Initialize(Ta[], int size, int ArraySize);
private:
    int CurrentSize, MaxSize;
    T *heap; //元素数组
};

```

---

最大堆的构造函数见程序 9-12。构造函数只是简单地开辟一个足够大的数组使之能够存储当元素个数达到最大值时的所有元素，它并不处理由 `new` 引发的 `NoMem` 异常。析构函数应删除此数组。`size` 函数尤其简单，仅返回 `CurrentSize` 的值。另一个简单的函数是 `Max`，如果最大堆为空，它将引发 `OutOfBounds` 异常；如果不为空，则返回最大树的根的值。

#### 程序 9-12 MaxHeap 的构造函数

```

template<class T>
MaxHeap<T> :: MaxHeap(int MaxHeapSize)
{ MaxSize=MaxHeapSize; heap=new T[MaxSize+1]; CurrentSize=0; }
//构造函数在 Insert（见程序 9-13）和 DeleteMax（见程序 9-14）的代码中假设已重载了关系
//操作符<、>和>=

```

---

最大堆的插入算法见程序 9-13。

#### 程序 9-13 最大堆的插入算法

```

template<class T>
MaxHeap<T>& MaxHeap<T> :: Insert(const T& x)
{ //把 x 插入到最大堆中
    if (CurrentSize==MaxSize) throw NoMem(); //没有足够空间
    //为 x 寻找应插入位置，i 从新的叶结点开始，并沿着树上升
    int i=++CurrentSize;
    while (i !=1 && x > heap[i/2]) { //不能够把 x 放入 heap[i]
        heap[i]=heap[i/2]; //将元素下移
        i /=2; //移向父结点
    }
    heap[i]=x;
    return *this;
}

```

---

在插入代码中，*i* 从新创建的叶结点位置 `CurrentSize` 开始，对从该位置到根的路径进行遍历。对于每个位置 *i*，都要检查是否到达根（*i*=1）或在 *i* 处插入新元素不会改变最大堆的性质（`x.key ≤ heap[i/2].key`）。只要这两个条件中有一个满足，就可在 *i* 处插入 *x*；否则，将执行 `while` 循环体，把位于 *i/2* 处的元素移到 *i* 处并把 *i* 处元素移到父结点（*i/2*）处。对于一

个具有  $n$  个元素的最大堆（即  $\text{CurrentSize}=n$ ），while 循环的执行次数为  $O(\text{height})=O(\lg n)$ ，且每次执行所需时间为  $O(1)$ ，因此 Insert 的时间复杂度为  $O(\lg n)$ 。

最大堆的删除算法见程序 9-14。

程序 9-14 最大堆的删除算法

```
template<class T>
MaxHeap<T>& MaxHeap<T>::DeleteMax(T& x)
{ //将最大元素放入 x，并从堆中删除最大元素，检查堆是否为空
    if (CurrentSize==0) throw OutOfBounds(); //队列空
    x=heap[1]; //最大元素
    //重构堆
    Ty=heap[CurrentSize--]; //最后一个元素
    //从根开始，为 y 寻找合适的位置
    int i=1; //堆的当前结点
    ci=2; //i 的孩子
    while (ci <=CurrentSize) { //heap[ci] 应是 i 的较大的孩子
        if (ci < CurrentSize && heap[ci] < heap[ci+1]) ci++;
        //能否把 y 放入 heap[i] 中
        if (y >=heap[ci]) break; //能
        heap[i]=heap[ci]; //不能，将孩子上移
        i=ci; //下移一层
        ci *=2;
    }
    heap[i]=y;
    return *this;
```

在 DeleteMax 操作中，堆的根（即最大元素）heap[1]被保存到变量 x 中，堆的最后一个元素 heap[CurrentSize]被保存到变量 y 中，堆的大小（CurrentSize）被减 1。在 while 循环中，开始查找一个合适的位置以便重新将 y 插入。从根部开始沿堆向下查找，对于具有  $n$  个元素的堆，while 循环的执行次数为  $O(\lg n)$ ，且每次执行所花时间为  $O(1)$ ，因此，DeleteMax 操作总的时间复杂度为  $O(\lg n)$ 。注意，即使堆的元素个数为 0，代码也能正确执行，在这种情况下不执行 while 循环，对堆的位置 1 进行赋值是多余的。

### 3. 堆调整——筛选

下面说明自堆顶至叶子的调整过程。例如，如图 9.8（a）所示是个最大堆，假设输出堆顶元素之后，以堆中最后一个元素替代之，如图 9.8（b）所示。此时为保证根结点的左、右子树均为最大堆，仅需将新的根结点自上至下进行调整即可，也就是完成一次筛选过程。首先，以堆顶元素和其左、右子树根结点的值比较之，由于右子树根结点的值大于左子树根结点的值且大于根的值，则交换 17 和 69；由于 17 替代了 69 之后破坏了右子树的最大堆，因此需进行和上述相同的调整，直至叶子结点调整后的状态如图 9.8（c）所示，此时堆顶为  $n-1$  个元素中的最大值。重复上述过程，将堆顶元素 69 和堆中最后一个元素 43\*进行交换且调整，得到如图 9.8（d）所示的新堆。

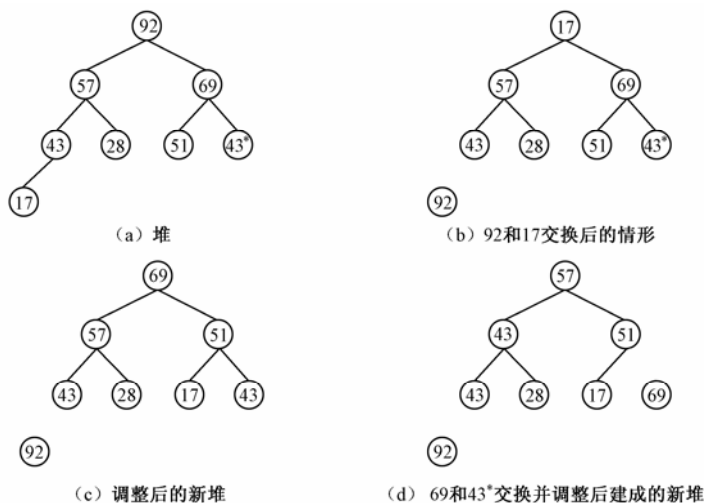


图 9.8 输出堆顶元素并调整建新堆的过程

堆的调整算法见程序 9-15。

程序 9-15 堆的调整算法

```
template<class T>
void HeapAdjust(SqList<T> &H, int s, int m) {
//已知 H.r[s..m]中记录的关键码除 H.r[s].key 之外均满足堆的定义
//本函数调整 H.r[s]的关键码, 使 H.r[s..m]成为一个大顶堆 (对其中记录的关键码而言)
RedType<T>rc=H.r[s];
    for (int j=2*s; j<=m; j*=2) { //沿 key 较大的孩子结点向下筛选
        if (j<m && LT(H.r[j].key, H.r[j + 1].key))++ j; //j 为 key 较大的记录的下标
        if (!LT(rc.key, H.r[j].key)) break; //rc 应插入在位置 s 上
        H.r[s]=H.r[j]; s=j;
    } //for
    H.r[s]=rc; //插入
} //HeapAdjust
```

#### 4. 建堆

为一个无序序列建堆的过程就是一个反复“筛选”的过程, 若将此序列看成是一棵完全二叉树, 则显然只需要对所有非叶子结点进行筛选 (因为叶子结点都满足堆的性质), 因此从最后一个非叶子结点 (即第 $\lfloor n/2 \rfloor$ 个元素) 开始进行筛选。

**【例 9-7】** 图 9.9 (a) 中的二叉树表示一个有 9 个元素的无序序列{48, 39, 67, 52, 16, 22, 17, 68, 24}。筛选从第 4 个元素开始, 由于  $52 < 68$ , 交换之, 交换后的序列如图 9.9 (b) 所示; 由于第 3 个元素 67 不小于其左、右子树根的值, 因此筛选后的序列不变, 而在第 2 个元素 39 被筛选之后序列的状态如图 9.9 (c) 所示; 图 9.9 (d) 所示为筛选根元素 48 之后建成的堆。



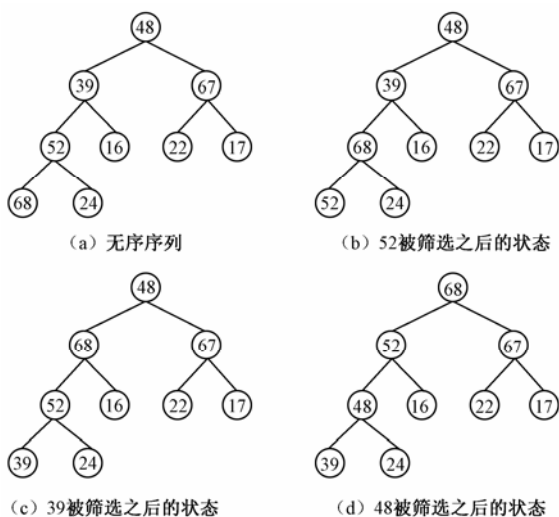


图 9.9 建堆的过程示例

## 5. 堆排序

堆排序的实质就是对无序序列不断进行建堆和调整堆的过程。为使排序后的记录序列按关键码非递减排列，在堆排序的实现过程中可先建一个最大堆，将堆顶记录与序列中最后一个记录进行交换，然后对序列中前  $n-1$  记录进行筛选，重新将它调整为一个最大堆，如此反复，直至排序结束。堆排序算法如程序 9-16 所示。

程序 9-16 堆排序算法

---

```

template<class T>
void HeapSort(SqlList<T> &H) {
//对顺序表 H 进行堆排序
    RedType<T> temp;
    for (int i=H.length/2; i>0; --i) //把 H.r[1..i]建成最大堆
        HeapAdjust(H, i, H.length);
    for (i=H.length; i>1; --i) {
        temp=H.r[1]; //将堆顶记录和当前未经排序子序列
        H.r[1]=H.r[i]; //H.r[1..i]中最后一个记录相互交换
        H.r[i]=temp; HeapAdjust(H, 1, i-1); //将 H.r[1..i-1]重新调整为最大堆
    } //for
} //HeapSort

```

---

当记录数比较少时，堆排序的效率并不理想，但当记录数很大时，堆排序是非常有效的。在最坏的情况下，堆排序的时间复杂度也为  $O(n \log n)$ ，同时堆排序的时间效率与待排序记录的初始次序无关，这是堆排序优于简单选择排序的地方，但堆排序也是不稳定排序，同时它需要一个  $O(1)$  的辅助空间。

## 9.5 归 并 排 序

归并排序 (merge sort) 是利用“归并”技术来进行排序的,而归并是指将两个或两个以上的有序表组合成一个新的有序表。算法思想是,将一个具有  $n$  个待排序记录的序列看成是  $n$  个长度为 1 的有序序列,然后进行两两归并,得到  $\lceil n/2 \rceil$  个长度为 2 的有序序列,再进行两两归并,得到  $\lceil n/4 \rceil$  个长度为 4 的有序序列,如此重复,直至得到一个长度为  $n$  的有序序列为止,这种排序方法称为 2 路归并排序。2 路归并排序的核心操作是将一维数组中前后相邻的两个有序序列归并为一个有序序列,其算法实现过程如程序 9-17 所示。

程序 9-17 2 路归并排序算法

---

```
template<class T>
void Merge(RedType<T> *SR, RedType<T> *&TR, int i, int m, int n) {
//将有序的 SR[i..m-1] 和 SR[m..n] 归并为有序的 TR[i..n]
    int j, k;
    for (j=m+1, k=i; i<=m && j<=n; ++k)
    { //将 SR 中记录由小到大并入 TR 中
        if (SR[i].key<= SR[j].key) TR[k]=SR[i++]; else TR[k]=SR[j++];
    } //for
    if (i<=m) { for (int il=k, jl=i; il<=n&&jl<=m; il++, jl++)
        TR[il]=SR[jl]; } //将剩余的 SR[i..m-1] 复制到 TR 中
    if (j<=n) { for (int il=k, jl=j; il<=n&&jl<=m; il++, jl++)
        TR[il]=SR[jl]; //将剩余的 SR[j..n] 复制到 TR 中
    }
} //Merge
```

---

一趟归并排序的操作是,调用  $\lceil n/(2h) \rceil$  次算法 merge,将  $SR[1..n]$  中前后相邻且长度为  $h$  的有序段进行两两归并,得到前后相邻、长度为  $2h$  的有序段,并存放在  $TR[1..n]$  中,整个归并排序需进行  $\lceil \log_2 n \rceil$  趟。可见,实现归并排序需要与待排记录等数量的辅助空间,其时间复杂度为  $O(n \log_2 n)$ 。递归形式的 2 路归并排序的算法如程序 9-18 所示。

程序 9-18 递归形式的 2 路归并排序的算法

---

```
template<class T>
void MSort1(RedType<T> *SR, RedType<T> *&TR1, RedType<T> *&TR2,
    int left, int right)
{ //将 SR[left..right] 和 SR[m+1..n] 归并排序为 TR1[left..right]
    int m;
    if (left==right) TR2[left]=SR[left];
    else { m=(left + right)/2; //将 SR[left..right] 平分为 SR[left..m] 和 SR[m+1..right]
        MSort1(SR, TR1, TR2, left, m); //递归地将 SR[left..m] 归并为有序的 TR2[left..m]
        MSort1(SR, TR1, TR2, m+1, right); //递归地将 SR[m+1..right] 归并为有序的 TR2[m+1..right]
        Merge(TR2, TR1, left, m, right);
        //递归地将 TR2[left..m] 和 TR2[m+1..right] 归并到 TR1[left..right] 中
    }
```

```

        for(int k=left; k<=right; k++) TR2[k]=TR1[k]; //同步更新中间变量 TR2
    } //else
} //MSort1

```

## 程序 9-19 归并排序算法

```

template<class T>
void MergeSort(SqlList<T> &L)  { //对顺序表 L 进行归并排序
    RedType<T> *rr=L.r; RedType<T> *TR2=new RedType<T>[L.length+1];
    MSort1(L.r, rr, TR2, 1, L.length);
    delete TR2[ ];
} //MergeSort

```

**【例 9-8】** 如图 9.10 所示为 2 路归并排序的一个例子。

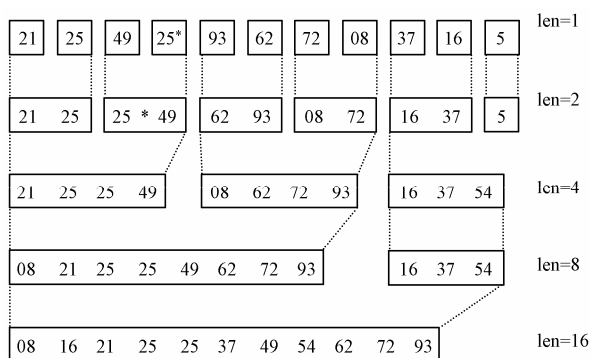


图 9.10 2 路归并排序示例

## 9.6 基数排序

基数排序 (radix sorting) 借助多关键码排序的思想, 采用“分配”与“收集”的方法对单逻辑关键码进行排序。基数排序不需要进行记录关键码间的比较。

### 9.6.1 多关键码的排序

#### 1. 定义

假设有  $n$  个记录的序列

$$\{R_1, R_2, \dots, R_n\}$$

且每个记录  $R_i$  中含有  $d$  个关键码 ( $K_i^0, K_i^1, \dots, K_i^{d-1}$ ), 则称以上序列对关键码 ( $K_i^0, K_i^1, \dots, K_i^{d-1}$ ) 有序是指, 对于序列中任意两个记录  $R_i$  和  $R_j$  ( $1 \leq i < j \leq n$ ) 都满足下列有序关系: ( $K_i^0, K_i^1, \dots, K_i^{d-1}$ ) < ( $K_j^0, K_j^1, \dots, K_j^{d-1}$ ), 其中,  $K_i^0$  称为最主位关键码,  $K_i^{d-1}$  称为最次位关键码。

## 2. 多关键码排序的实现

### (1) 最高位优先 MSD (most significant digit first)

#### ① 算法思路

先对最主位关键码  $K^0$  进行排序, 将序列分成若干子序列, 每个子序列中的记录都具有相同的  $K^0$  值, 然后分别就每个子序列对关键码  $K^1$  进行排序, 按  $K^1$  值不同再分成若干更小的子序列, 如此重复, 直至对  $K^{d-2}$  进行排序之后得到的每一子序列中的记录都具有系统的关键码( $K^0, K^1, \dots, K^{d-2}$ ), 而后每个子序列分别对  $K^{d-1}$  进行排序, 最后将所有子序列依次连接在一起成为一个有序序列为止。

#### ② 特点

按 MSD 进行排序时, 必须将序列逐层分割成若干子序列, 然后对各子序列分别进行排序。

### (2) 最低位优先 LSD (least significant digit first)

#### ① 算法思路

从最次位关键码  $K^{d-1}$  开始排序, 然后再对高一位的关键码  $K^{d-2}$  进行排序, 如此重复, 直至对  $K^0$  进行排序后便成为一个有序序列为止。

#### ② 特点

i) 按 LSD 进行排序时, 不必分成子序列, 对每个关键码都是整个序列参加排序, 但对  $K^i (0 \leq i \leq d-2)$  进行排序时, 只能用稳定的排序方法。

ii) 按 LSD 进行排序时, 在一定条件下 (即对前一个关键码  $K^i (0 \leq i \leq d-2)$  的不同值, 后一个关键码  $K^{i+1}$  均取相同值), 可通过若干次“分配”和“收集”来实现排序。

**【例 9-9】** 扑克牌中每张牌都具有两个关键码: “花色”和“面值”, 若规定扑克牌中“花色”的大小次序关系为: “梅花” < “方片” < “红桃” < “黑桃”, “面值”的大小次序关系为: “2” < “3” < ... < “A”, 且“花色”的地位高于“面值”, 则扑克牌可按上述两种不同的排序方法整理成有序序列。

① MSD: 先按不同“花色”分成有次序的 4 堆, 每一堆的牌均具有相同的“花色”, 然后分别对每一堆按“面值”大小整理有序。

② LSD: 先按不同“面值”分成 13 堆, 然后将这 13 堆牌自小至大叠在一起 (“3”在“2”之上, “4”在“3”之上, …… , 最上面的是 4 张“A”), 然后将这副牌整个颠倒过来再重新按不同“花色”分成 4 堆, 最后将这 4 堆牌按自小至大的次序合在一起 (这时, 梅花在最下面, 黑桃在最上面)。

## 9.6.2 链式基数排序

### 1. 算法思想

有的逻辑关键码可看成由若干个关键码复合而成, 且每个关键码  $K^i$  都在相同的范围内, 则按 LSD 进行排序时, 只要从最低数位关键码开始, 按关键码的不同值将序列中的记录“分配”到 RADIX 个队列中后再进行“收集”, 如此重复  $d$  次, 按这种方法实现的排序称为基数排序, 其中, “基”指的是 RADIX 的取值范围。而链式基数排序就是用链表作存储结构的基数排序。例如, 若关键码是数值, 且其值都在  $0 \leq K \leq 999$  范围内, 则可把每一

个十进制数字看成一个关键码，即认为  $K$  由 3 个关键码 ( $K^0, K^1, K^2$ ) 组成，其中  $K^0$  是百位数， $K^1$  是十位数， $K^2$  是个位数，对每一个关键码  $0 \leq K^i \leq 9$  ( $i=0, 1, 2$ )，“基”为 10。基数排序算法的实现参见图 9.11。

**【例 9-10】** 如图 9.11 所示为链式基数排序三趟“分配”与“收集”的过程示例。

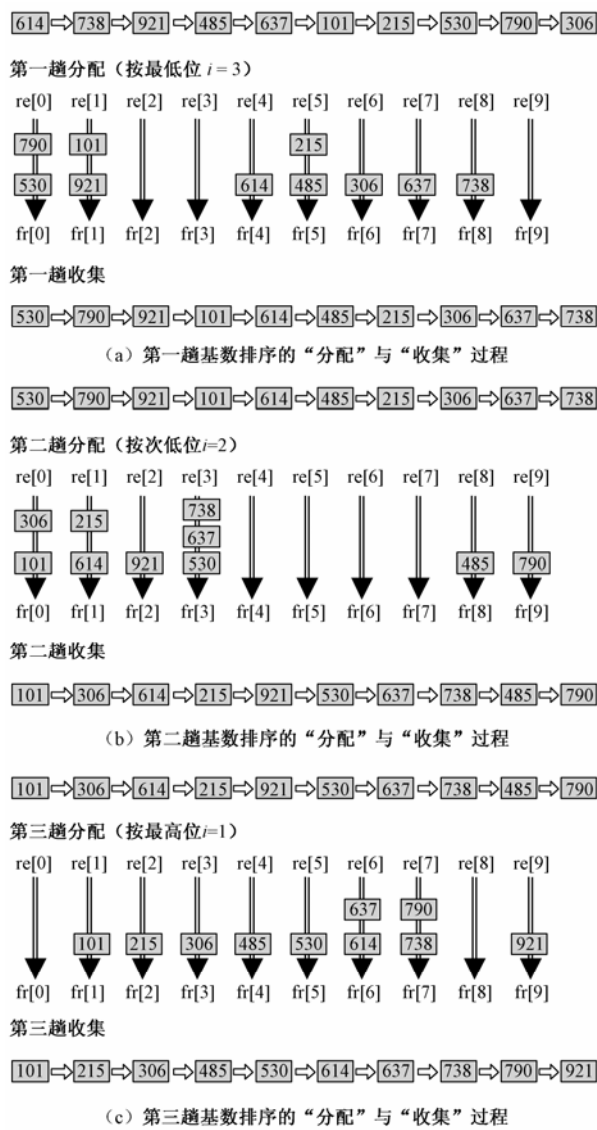


图 9.11 链式基数排序示例

## 9.7 内排序方法的比较和讨论

### 1. 排序方法的比较

综合上面的讨论，表 9.1 中给出了本章所介绍的排序算法的性能比较情况。

表 9.1 各种排序方法的性能比较表

排 序 方 法	时间复杂度 (平均情况)	特 殊 情 况	辅助存储空间	稳 定 性
直接插入排序	$O(n^2)$	初始序列有序 $O(n)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
冒泡排序	$O(n^2)$	初始序列有序 $O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1+\epsilon})$ , $\epsilon$ 是介于 0 和 1 之间的 常数, 即 $0 < \epsilon < 1$	无	$O(1)$	不稳定
快速排序	$O(n \log_2 n)$	初始序列有序 $O(n^2)$	$O(1)$	不稳定
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定
2 路归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	稳定
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(1)$	稳定

2. 影响排序效果的因素

不同的排序方法适应不同的应用环境和要求，因此在选择合适的排序方法时应综合考虑下列因素：待排序的记录数目  $n$ ，记录的大小（规模），关键码的结构及其初始状态，对稳定性的要求，语言工具的条件，存储结构，时间和辅助空间复杂度等。经证明，借助于“比较”进行排序的算法在最坏情况下能达到的最好的时间复杂度为  $O(n \log n)$ 。

由此可知，本章讨论的所有排序方法中没有哪一种是绝对最优的，在实际应用中应根据不同情况进行选用，甚至可将多种方法结合起来使用。

本 章 总 结

1. 学习要点

本章主要介绍了各种内部排序（插入排序、快速排序、选择排序、归并排序、基数排序）方法的特点和实现算法，以及各种排序方法在时间复杂度方面的讨论和比较，主要学习要点如下：

- ① 内排序方法的基本思想及稳定和非稳定排序的区别；
- ② 各种内排序算法及其优缺点；
- ③ 各种内部排序算法的时空性能和适用场合。

2. 基本要求

(1) 深刻理解排序的基本概念和各种排序方法的算法思想及特点

- ① 清楚排序的定义，稳定与不稳定排序、内排序与外排序的概念及区别；
- ② 知道各种排序的实质是以关键码的比较为基础的；
- ③ 知道各种内部排序（插入、快速、选择、归并、基数等排序）方法的主要优缺点及适用场合。

(2) 掌握插入排序、快速排序和选择排序的基本思想和算法

- ① 掌握快速排序的基本思想，一趟快速排序交换过程及每趟快速排序的结果；

- ② 掌握直接插入排序算法的基本思想及时空性能指标;
- ③ 知道快速排序算法的时空性能及交换排序的指导思想;
- ④ 知道直接选择排序的基本思想及其性质;
- ⑤ 掌握堆排序及归并排序和外部排序的基本思想和实现算法;
- ⑥ 掌握堆排序的表示方法、调整方法和“筛选”过程;
- ⑦ 掌握建堆方法、堆排序算法的时空性能;
- ⑧ 掌握归并排序的指导思想和将两个有序文件合并成一个有序文件的算法;
- ⑨ 知道 2 路归并排序的基本思想和时空性能。

### 3. 重点与难点

本章的重点是：快速排序和直接插入排序及堆排序的实现算法。难点是：二叉排序树和堆排序算法及时空性能分析。

## 习题 9

- 9-1 修改气泡排序算法并写出程序，使每趟排序后最大元素排在最后，下趟参加排序的元素则减少一个。
- 9-2 给出一组关键码 {12, 2, 16, 30, 8, 4, 10, 20, 6, 18}，写出按降序排序时，用以下方法排序的每一趟排序的结束状态：(1) 折半插入排序；(2) 快速排序；(3) 堆排序；(4) 基数排序。
- 9-3 写出对习题 9-2 中给出的一组数据用归并排序的方法按升序排序的算法或程序。
- 9-4 用链表表示两组有序数据，写出归并排序的算法。
- 9-5 用哈希法排列一组字符串数据，要求用每个字符串的第一个字母在字母表中的位置（如 A 为 1，B 为 2 等），作关键码组成哈希函数  $h(k_i) = \lfloor V \times M/26 \rfloor$ ，其值为相应元素在哈希表中的位置。其中， $V$  为字符串的关键码， $M$  为哈希表的长度。规定用线性探测再散列的方法解决冲突。现要求写出程序完成以下功能：① 建立哈希表，并输出此表；② 计算找到某个数据元素需要检测的次数以及对所有数据检测的次数。
- 9-6 某系学生举行英语竞赛，共 500 人参加，竞赛分三部分进行：第一部分为文法，第二部分为笔译，第三部分为口语。竞赛结果的名次按下述原则排列：首先看总分的高低，总分相同者看口语成绩的高低，总分和口语相同者看笔译成绩的高低。设计一个结构，存放学生的竞赛结果，然后对所有学生输出各人的名次，每个学生的信息应包含以下数据项：学生姓名、文法成绩、笔译成绩、口语成绩和总分。
- (1) 要求以下列两种方式输出：① 按名次顺序将所有学生信息输出；② 按姓名的汉语拼音字母顺序输出所有学生信息，并列出学生竞赛的名次。(2) 用折半检索法，检索指定学生的成绩、总分及名次。若找不到，应给出提示信息。
- 9-7 有 12 个整数：23, 37, 7, 79, 29, 43, 73, 19, 31, 61, 23, 47，按下列方法将这组整数排序，并分别写出每遍处理后的数列：① 冒泡排序；② 插入排序；③ 选择排序；④ 快速排序；⑤ 堆排序；⑥ 归并排序；⑦ 基数排序；⑧ 希尔排序。
- 9-8 证明下述排序过程是稳定排序：① 冒泡排序；② 插入排序；③ 选择排序；④ 2 路归并排序。
- 9-9 假定被排序的数据集合用一个单向链表表示，其表头结点指针为  $f$ ，结点的类型定义参见程序 2-8 中的定义。分别按下列算法写出排序过程  $\text{sort}(F)$ ：① 插入排序；② 选择排序。
- 9-10 假定学生成绩表中数据的类型定义如下：

```
typedef struct { nametype name;
                float grade;
```

```
int place
}score;
```

其中, `grade` 表示考试成绩, `place` 表示按成绩排列的名次。数组 `a` 为 `score` 型变量, 并假定 `a[1], …, a[n]` 表示一个班的学生成绩表。写一个过程 `sting(n, a)`, 在学生成绩表中填入每个学生按成绩排列的名次, 并使成绩相同者并列同一名次。这样的排序方法称为计数排序, 并分析计数排序的时间复杂度。

9-11 假定数据集中的每个元素都是字符行, 每行的长度均为 `length`, `line` 是每行的指针构成的数组, 试设计一个过程 `sort(n, line)`, 采用希尔排序, 把 `line` 中的行指针按照字符行的字典顺序排列。

9-12 在堆排序算法中, 通过语句 `for (i=n div 2 downto 1) do pushdown(i, n)` 调用过程 `pushdown` 建立初始的堆。(1) 按照自己选定的一组数据, 画出每次调用 `pushdown` 之后形成的堆结构图。(2) 试分析完成建初始堆所需的时间。

9-13 设每个元素关键码是由小写字母组成的字符串, 其长度为 10。写出对这种类型的数据进行基数分类的过程。

9-14 随机地设置 100 个 4 位的整型数存入计算机中。分别按照插入排序、Shell 排序、快速排序、归并排序、堆排序、基数排序方法, 开发实现程序。然后对同一组数据进行分类, 比较执行每个程序时计算 CPU 运行的时间。

9-15 对  $n$  个元素组成的线性表进行快速排序时需要进行的比较次数依赖于这  $n$  个元素的初始排列, 试问: ①  $n=7$  时, 在最好情况下需要进行多少次比较? 请说明理由。②对  $n=7$  给出一个最好情况的初始排列实例。

9-16 一个线性表元素由正整数和负整数组成, 利用一趟快速排序的思想编写一个算法, 把正整数和负整数分开, 使线性表前半为负整数, 后半为正整数 (提示: 利用一趟快排进行查找的思想)。

9-17 已知  $(k_1, k_2, \dots, k_n)$  是堆, 试写一个算法将  $(k_1, k_2, \dots, k_n, k_{n+1})$  调整为堆。由此思想写一个从空堆开始一个一个添入元素的建堆算法 (提示: 增加一个  $k_{n+1}$  后, 应从叶子向根的方向进行调整)。

9-18 (1) 折半插入算法每次循环时, 记录是否保持具有相同关键码值记录的原始相对次序? (2) 试修改冒泡算法为摇动分类法, 即交替地从正、反两个方向进行扫描, 第一次把关键码值最大的记录放到最末尾, 如此反复进行。

9-19 编写一个 C++ 语言程序, 打印出 6 个正整数  $a_1, a_2, a_3, a_4, a_5$  和  $a_6$  的所有集合, 使之满足下列条件:  $a_1 \leq a_2 \leq a_3 \leq 20, a_1 < a_4 \leq a_5 \leq a_6 \leq 20$ , 并使得  $a_1, a_2$  和  $a_3$  的平方和等于  $a_4, a_5$  和  $a_6$  的平方和。

9-20 试构造一个排序算法使 5 个整数最多采用 17 次比较。

9-21 设计一个用链表表示的选择排序算法的程序。

9-22 (1) 冒泡排序在什么情况下, 序列会向与排序相反的方向移动, 试举例说明之。快速排序有这种现象吗? (2) 写一个非递归的快速排序方法的过程。

9-23 用堆排序算法对具有下列关键码值的结点进行排序, 试描述每次调用函数 `shift` 后的堆结构及其结点的关系。



# 第 10 章 文件组织和外排序

尽管数据管理技术早已从文件系统发展到数据库系统，但因为文件系统是数据库系统的基础，从专用、高效和系统软件开发的角度来看，文件系统仍有其不可取代的地位。在数据处理方面，特别是事务型的软件编制工作，都涉及有关文件的知识。通常，将存储在主存储器（内存）中的记录集合称为表，存储在辅助存储器（外存）中的记录集合称为文件。如何有效地组织文件中的数据，提供方便而高效的文件数据读取方法，是本章所要讨论的内容。

## 10.1 外存储器概述

目前广泛使用的外存储器有磁带机和磁盘机两种，前者为顺序存取的存储设备，后者为直接存取的存储设备。

### 10.1.1 磁带及其信息的存取

磁带是涂上一层薄薄磁性材料的一条窄带。使用时，将磁带盘放在磁带机上，驱动器控制磁带盘转动，带动磁带向前移动，通过读/写头读出磁带上的信息或者将信息写入磁带。磁带是一种启停设备，它可根据读/写需要随时启动和停止。由于读/写信息应在旋转稳定状态下进行，而磁带从启动到稳定旋转或从旋转到静止都需要一个“启停时间”（即加速或减速的过渡时间），为了适应启停时间，信息在磁带上不能连续存放，而要在相邻两个“字符”之间留出一定长度（通常为 1/4 英寸~1/3 英寸）的空白区，称为间隙 IRG（inter record gap），也就是磁带上相邻两组字符组（记录）之间的空白区。两个间隙之间的字符组称为一个“物理记录”或者“页块”。页块是内外存信息交换的单位。内存中用来暂时存放一个页块的区域称为“缓冲区”。例如，图 10.1（a）和（b）分别表示信息以字符为单位和以块为单位在磁带上进行存储的情况。

成块的优点：①可减少 IRG 的数目，从而提高磁带的利用率，块的长度大于 IRG 的长度。②可减少 I/O 操作。一次 I/O 操作就可把整个物理块都读到内存缓冲区中，然后再从缓冲区中取出所需要的信息（一个字符组）。每当要读一个字符组时，首先检查缓冲区中是否已有该字符组，若有，则不必执行 I/O 操作，直接从缓冲区中读取即可。

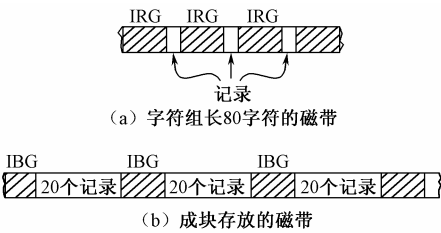


图 10.1 磁带上信息存放示意图

磁带上读/写一块信息所需的时间为  $T_{I/O} = t_a + n t_w$ 。其中,  $t_a$  为延迟时间, 即读/写头到达传输信息所在物理块起始位置所需时间 (显然, 延迟时间与信息在磁带上的位置、当前读/写头所在位置有关);  $t_w$  为传输一个字符的时间。

### 10.1.2 磁盘及其信息的存取

磁盘是一种直接存取的存储设备, 既能顺序存取, 又能随机存取。目前使用较为广泛的是磁头可移动的活动的磁头磁盘。通常, 若干盘片组成一个盘片组固定在一个主轴上, 随着主轴按一个方向高速旋转, 除最上面和最下面的两个外侧盘面外, 其余用于存储数据的盘面称为“记录盘面”, 简称“记录面”, 记录面上存储数据的同心圆称为“磁道”。每个记录面上有一个读/写磁头, 所有读/写头安装在一个活动臂装置上, 可一起做径向移动。当磁道在读/写头下通过时, 便可进行信息的读/写。

各记录盘面上直径相同的磁道组成一个“柱面”, 一个磁道又可分为若干弧段, 称为“扇面”。磁盘信息存取的单位为一个扇面的字符组, 称为一个“页块”。因此, 通常用一个三维地址来表明磁盘信息: 柱面号、记录面号和页块号。为了访问一块信息, 首先必须移动活动臂使磁头移到所需柱面 (称为定位或寻查), 然后等待页块起始位置转到读/写头下, 最后读/写所需信息, 所需时间由这三个动作所需的时间组成

$$T_{I/O} = t_{seek} + t_{la} + n t_{wm}$$

式中,  $t_{seek}$  为寻查时间;  $t_{la}$  为等待时间, 即等待信息块的初始位置旋到读/写头下的时间;  $t_{wm}$  为传输 (一个字符的) 时间;  $n$  为页块内字符数目。

磁盘又分为两类, 一类是硬盘, 另一类是软盘。

软盘 (磁盘) 盘片是一种圆形盘片, 以软质的塑料薄片为载体, 上面涂敷磁性材料作为记录介质。软盘的大小有 8 英寸、5.25 英寸、3.5 英寸等几种。一般软盘都有一个塑料外壳, 较坚硬, 它的作用是保护里边的盘片不被划伤, 并还能保持盘片的清洁度。另外, 软盘还提供了一个写保护口, 若写保护口关闭, 则只能从盘中读数据, 而不能将数据写入盘中; 若写保护口打开, 则既可从盘中读数据, 也可往盘中写入数据。写保护是个非常有用的功能, 可防止误写操作, 也可避免病毒对它的侵害。新的软盘在使用前要进行格式化。

硬盘包括一组刚性的、圆盘状的盘片, 它们通常由铝或玻璃制成。与软盘不同的是, 这些盘片不能弯曲或折叠。硬盘是目前计算机使用的主要存储设备, 主要因为它的访问速度比软盘和光盘都快, 因此它最适合于储存那些需要经常访问和需要快速访问的程序和文件。计算机的操作系统、应用软件和重要数据资料都存储在硬盘之中, 因此要求硬盘的容量尽可能大, 速度尽可能快, 并且安全性也较高。

### 10.1.3 U盘

U 盘 (也称优盘、闪盘) 是一种可移动的数据存储工具, 具有容量大、读/写速度快、体积小、携带方便等特点。插入任何计算机的 USB 接口都可直接使用。同时, 它还具备防磁、防震、防潮的诸多特点, 明显增强了数据的安全性。U 盘的性能稳定, 数据传输高速高效, 较强的抗震性能可使数据传输不受干扰, 并且价格适中, 容量和速度也远胜于软盘, 已成为软盘的替代品。

## 10.2 文件的基本概念

### 10.2.1 文件

#### 1. 基本概念

文件：性质相同的记录的集合。

数据项：最基本的不可分的数据单位，是文件中可使用的数据的最小单位。

属性：记录中所有非关键字的数据项，称为记录的属性。

#### 2. 文件的分类

(1) 按记录类型的不同

① 操作系统文件：操作系统中的文件仅是一维的连续的字符序列，无结构、无解释。

② 数据库文件：数据库中的文件是带有结构的记录的集合。这类记录是由一个或多个数据项组成的集合，它也是文件中可存取的数据的基本单位。其中，数据库文件按记录中关键字的多少又可分为：

- 单关键字文件——文件中的记录只有一个唯一标识记录的主关键字；
- 多关键字文件——文件中的记录除了含有一个主关键字外，还含有若干个次关键字。

(2) 按记录含有信息长度的不同

① 定长记录文件：文件中每个记录含有的信息长度相同。

② 不定长记录文件：文件中每个记录含有的信息长度不等。

#### 3. 记录的逻辑结构和物理结构

文件的逻辑特性是由文件中各个记录按照某种次序排列而形成的某种逻辑关系确定的，记录的逻辑结构同时也反映了用户对数据的表示和存取方式，主要着眼于用户使用的方便性。而记录的物理结构是数据在物理存储器上存储的方式，是数据的物理表示和组织，它着眼于提高存储空间的利用率和减少存取记录的时间。

### 10.2.2 文件的操作（运算）与存取

#### 1. 文件的操作

文件的操作主要包括对记录的存取（读写）、检索和修改（插入、删除或更新记录）。其中，对记录的存取可分为：① 顺序存取，即根据记录序号存取下一个逻辑记录；② 直接存取，即存取记录序号中的第  $i$  个逻辑记录；③ 按关键字存取，即给定一个值，查询一个或一批关键字与给定值相关的记录。

对数据文件的查询可有 4 种方式：① 简单询问，基于给定的关键码值或某属性值查询；② 范围询问，基于查询关键码值或某属性值的范围查询；③ 函数询问，基于给定关键码值的某个函数查询；④ 布尔询问，基于以上 3 种查询条件的布尔组合查询。

文件的操作有实时和批量两种不同方式。通常，实时处理对应答时间要求严格，应在

接受询问之后几秒内完成检索和修改，而批量的处理则不然。例如，银行的账户系统需要完成实时检索，但可进行批量修改，可将一天的存款和提款信息记录在一个事务文件中，在一天的营业之后再进行处理。

## 2. 文件的存取方式

(1) 顺序存取文件 (sequential access file)，简称“顺序文件”。数据写入驱动器的方式是按照数据输入的先后次序进行存放的，而数据的读取方式也是按照其输入顺序读取的，即从第一个记录开始读取。在这种数据文件中，每一个记录的长度可能不一样，虽然比较节约空间，但由于每次查询都必须从头开始进行顺序查找，因此，越晚输入的数据的查找时间越长。

(2) 随机存取文件 (random access file)，简称“随机文件”。每一个记录在磁盘中所占的长度都相同，数据存入磁盘的方式没有先后次序的限制。由于每个数据占用的长度固定，查询时可利用公式计算出该数据的位置以便进行快速存取。不管数据在文件中的位置如何，其查询时间大致相同。每个数据所占磁盘空间的大小，以记录长度最长的那个为基准，因此，当每个数据实际长度差异很大时，采用随机文件会比较浪费磁盘空间。

### 10.2.3 文件的物理结构

文件的物理结构即为文件在存储介质（磁盘或磁带）上的组织方式，文件的基本组织方式包括以下 4 种。

#### (1) 顺序组织

记录在外存储器中的存放顺序与记录在文件中的逻辑顺序完全一致，称按这种存储方式组织的文件为“顺序文件”。它是一种顺序组织方式，适于进行顺序存取（即存取一个记录之后接着存取其后继记录）和批量处理，但对顺序文件中记录的随机存取效率很低。

#### (2) 随机组织（散列/直接存取）

散列文件类似于哈希表，即根据文件中关键字的特点设计一种哈希函数（也称为散列函数）和处理冲突的方法来确定记录的存储位置，将记录散列在存储介质上，这样的文件被称作“散列文件”或“直接存取文件”。散列文件是一种随机组织方式，对于关键字值等于给定值的记录的访问，可直接由散列函数及冲突处理方法求得在外存上的存储位置，因此随机存取效率很高。但散列文件不适宜顺序存取和成批处理。

#### (3) 索引结构

索引文件是指除了文件本身之外，还为文件建立一个索引表，索引表的每一项都由一个关键字值和一个指针（即存储位置）构成的二元组  $(k, p)$  组成，其中， $k$  是对应记录的关键字值， $p$  是该记录的外存地址。每个索引项可对应文件的一个逻辑记录，叫密集索引。如果索引文件的记录按关键字排列有序，则称为索引顺序文件。对于索引顺序文件，可对一组记录建立一个索引项，这种索引叫作稀疏索引，此时  $k$  是一组记录上关键字值最小或最大记录的关键字值， $p$  是一组记录的外存地址。

对索引文件的检索过程分两步进行：首先查找索引表，若索引表上存在该记录，则根据索引项的指针域访问外存上的对应记录；否则，表明外存中不存在该记录，也就不需要访问外存了。索引文件适合于随机存取，而索引顺序文件既适合于随机存取也适合于顺序存取。

(4) 链组织结构

链组织结构类似于线性表的链表存储结构，记录之间用指针进行相互链接，这里的指针通常指页块的物理地址。这种结构将逻辑上连续的文件分散存放在若干不连续的物理块中，每个物理块设有一个指针，指向其后续的物理块。只要指明文件的第一个块号，就可按链指针检索整个文件。这种结构的优点是文件长度可以动态变化，其缺点是不适合随机访问。

10.3 顺 序 文 件

顺序文件（sequential file）是指记录按存入文件的先后顺序存放到外存储器上，即逻辑顺序和物理顺序一致的文件。其中，逻辑上相邻的记录存储在物理上相邻位置的顺序文件称为连续文件；逻辑上相邻的记录通过指针链相互指向的顺序文件称为串联文件。

磁带是典型的顺序存取外存储器。存储在磁带上的文件只能是顺序文件，而对顺序文件的操作主要根据记录的序号或记录的相对位置来实现，主要包括以下几个方面：①对文件中任意位置的记录进行查找时，需按照顺序访问的方式进行；②如果文件中的记录是按任意次序排列的，则在文件的末尾插入新记录；③修改文件中某个记录或在文件中间插入新记录时，需要用另一条磁带复制整个文件，并在复制的过程中用修改后的记录代替原来的记录或者在复制文件中间插入新记录。为提高磁带文件的处理效率，可对磁带文件进行批处理过程，即，将所有修改请求存入一个事务文件中并对该事务文件进行排序，将包含待修改的原始文件的主文件和事务文件归并成一个新的主文件。

磁盘作为外存储器时，既可进行顺序存取，又可进行随机存取。对于未排序的文件，只能按顺序方法存取；而对于记录有序的文件，可通过折半查找或内插查找法进行存取。磁盘上顺序文件的批处理和磁带文件类似，只是当修改项中没有插入操作，且更新时不增加记录的长度时，可不建立新的主文件，而直接修改原来的主文件即可。显然，磁盘文件的批处理可在一台磁盘组上进行。

10.4 索 引 文 件

(1) 索引表及相关术语

按关键字对文件记录进行存取时，需要用关键字在文件中进行查找，这样往往会因大量数据的输入和输出而降低查找的速度。为此，可建一个表，该表存储关键字值和相应记录的存储地址之间的对应关系，对该表进行查找就可提高效率，这种表称之为索引表。索引表示例如图 10.2 所示。其中，表中的每一项称为索引项，通常按关键字（或逻辑记录号）顺序排列。

关键字 $K_i$	物理地址（指针）
036	142
085	139
137	140
169	147
250	143
284	138
361	144

图 10.2 索引表示例

具有索引的文件称为索引文件，包括文件数据区和索引表两大部分。若该文件数据区中的记录按关键字顺序排列，则为索引顺序文件，否则为索引非顺序文件。当数据文件中记录不按关键字顺序排列时，必须对每个记录都建立一个索引项，如此建立的索引表称为稠密索引；当数据文件中的记录按关键字顺序有序时，可对一组记录建立一个索引项，这种索引表称为非稠密索引。

(2) 索引文件的检索

在索引文件中根据关键字值进行检索时，分两步进行，由于索引文件通常比较小，因此可一次读入内存。首先将外存中的索引块调入内存进行查找，若索引表上存在该记录，则再通过索引项的指示读取外存上的对应记录；否则，因为该记录不存在而不需要访问外存。因此，对索引文件的检索只需访问外存两次，一次读索引，一次读记录。索引表是有序的，在索引表中进行查找时还可利用折半查找法。

(3) 索引文件的修改

在索引表中删除一个记录时，仅需删去表中相应的索引项；在索引表中插入一个记录时，首先将记录存放到数据区的末尾，然后再在索引表中插入相应的索引项；在索引表中更新一个记录时，将更新后的记录置于数据区的末尾，同时修改索引表中相应的索引项。

(4) 多级索引

当文件的记录数目很多时，索引文件会很大而导致一个存储块放不下整个索引文件，这时，就需要建立一个索引的索引，即查找表。每个索引块在查找表中占一个项，用来指示索引块内最后一个索引项的关键字值和该块的地址，假设图 10.2 所示的索引表需要占 4 个物理块的外存，每一个物理块容纳 2 个索引，则建立的查找表如图 10.3 所示。

关键字	块号
085	1
169	2
284	3
361	4

图 10.3 图 10.2 中索引表的索引

通常，最高可有 4 级索引：数据文件→索引表→查找表→第二查找表→第三查找表。上述的多级索引是一种静态索引，各级索引均为顺序表结构。虽然顺序表的结构简单，但不便于修改，每次修改时都要重组索引，因此当数据文件在使用过程中记录变动较多时，应采用动态索引，如二叉排序树（或二叉平衡树）、B-树及键树等。

# 10.5 ISAM文件和VSAM文件

## 10.5.1 ISAM文件

索引文件分为数据区和索引表两部分，索引表中的索引项通常按关键码（或逻辑记录号）的顺序排列，若数据区中的记录按关键码顺序排列，则索引文件称为索引顺序文件；若数据区中的记录不按关键码顺序排列，则索引文件称为索引非顺序文件。下面将介绍两种常用的索引顺序文件：ISAM 文件和 VASM 文件。

1. ISAM文件组织

索引顺序存取方法（indexed sequential access method, ISAM）是一种专为磁盘存取设计的文件组织方式，采用静态索引结构。由于磁盘是以盘组、柱面和磁道三级地址存取的设

备，因此，可对磁盘上的数据文件建立盘组、柱面和磁道多级索引。ISAM 文件由多级主索引、柱面索引、磁道索引和主文件组成。如图 10.4 所示为存放一个磁盘组上的 ISAM 文件，每个柱面建立一个磁道索引，每个磁道索引项由基本索引项和溢出索引项两部分组成，每一部分都包括关键字和指针两项（见图 10.5）。

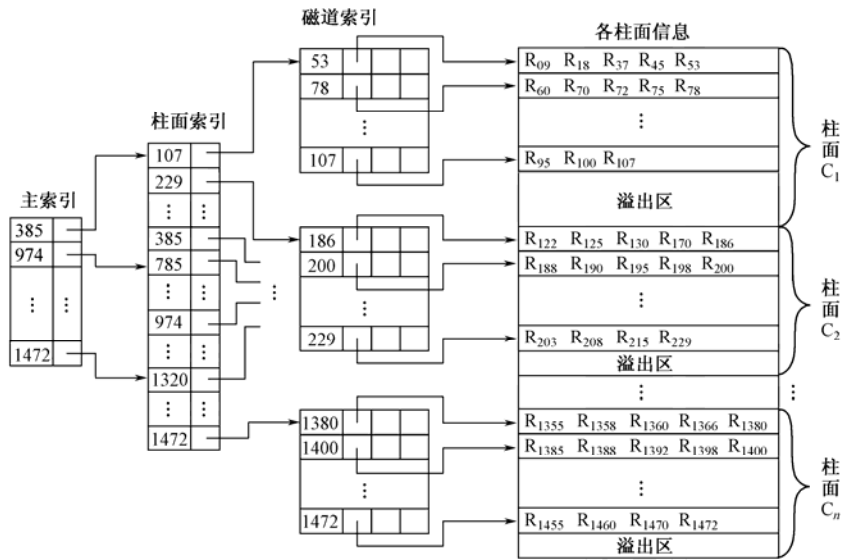


图 10.4 ISAM 文件结构示例

(1) 磁道索引项

基本索引项的关键字项表示该磁道中最末一个记录的关键字（在此为最大关键字），指针项指示该磁道中第一个记录的位置；溢出索引项的关键字项表示该磁道溢出的记录的最大关键字，指针项指示在溢出区中的第一个记录。

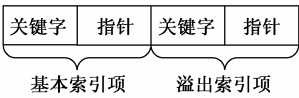


图 10.5 磁道索引项结构

(2) 柱面索引项

其关键字项表示该柱面中最末一个记录的关键字（最大关键字），指针项指示该柱面上的磁道索引位置。文件记录进行存储时，先放到一个柱面上，然后再顺序放到相邻的柱面上，在同一个柱面中，文件按盘面的次序进行顺序存放。柱面索引存放在某个柱面上，若柱面索引较大而占多个磁道时，则可建立柱面索引的索引——主索引。通常，主索引和柱面索引放在同一个柱面上。

2. ISAM操作

(1) ISAM 文件的检索

在 ISAM 文件中检索记录时，先从主索引出发，找到相应的柱面索引，再从柱面索引找到记录所在柱面的磁道索引，最后从磁道索引找到记录所在磁道的起始位置，由此出发在该磁道上进行顺序查找，直至找到为止；若找遍该磁道不存在此记录，则表明该文件中无此记录。若被查找的记录在溢出区，则可从磁道索引项的溢出索引项中得到溢出链表的头指针，然后对该表进行顺序查找。

### (2) 溢出区和插入操作

在 ISAM 文件中的每个柱面上开辟一个溢出区，并且磁道索引中有溢出区索引项，这是为插入记录设置的。每个柱面的基本区为顺序存储结构，而溢出区则是链表结构，同一磁道溢出的记录由指针相链。通常，溢出区可有三种设置方法：①集中存放，为整个文件设置一个大的单一的溢出区；②分散存放，为每个柱面设置一个溢出区，图 10.4 即为此设置法；③集中与分散相结合，溢出时，记录先移至每个柱面各自的溢出区，待满之后再使用公共溢出区。

### (3) 删除操作

在 ISAM 文件中，删除记录比插入记录简单得多，只需找到待删除的记录，在其存储位置上做删除标记即可，而不需要移动记录或改变指针。但在经过多次的增删后，文件的结构可能变得很不合理。此时，大量的记录进入溢出区，而基本区会有很大空间的浪费。因此，通常需要周期地整理 ISAM 文件，把记录读入内存，重新排列，复制成一个新的 ISAM 文件，填满基本区而空出溢出区。

## 10.5.2 VSAM文件

### 1. VSAM方法

虚拟存储存取方法（virtual storage access method, VSAM）是一种索引顺序文件的组织方式，采用  $B^+$  树作为动态索引结构，该方法利用了操作系统的虚拟存储器的功能，给用户方便。对用户来说，文件只有控制区间和控制区域等逻辑存储单位，与外存储器中的柱面、磁道等具体存储单位没有必然的联系。用户在存取文件中的记录时，不需要考虑这个记录的当前位置是否在内存，也无须考虑何时执行“读/写”操作。其中，控制区间（control interval）是一个 I/O 操作的基本单位，由一组连续的存储单元组成，同一文件上控制区间的大小相同，每个控制区间可视为一个逻辑磁道，而每个控制区域可视为一个逻辑柱面。控制区域使顺序集中一个结点连同对应所有控制区间形成的一个整体。

VSAM 文件的结构如图 10.6 所示。它由 3 部分组成：索引集、顺序集和数据集。文件的记录均存放在数据中。顺序集和索引集一起构成一棵  $B^+$  树，为文件的索引部分。顺序集中存放每个控制区间的索引项。每个控制区间的索引项由两部分信息组成，即该控制区间中最大关键字和指向控制区间的指针。若干相邻控制区间的索引项形成顺序集中的一个结点，结点之间用指针相链，而每个结点又在其上一层的结点中建有索引，且逐层向上建立索引。所有的索引项都由最大关键字和指针两部分信息组成，这些高层的索引项形成  $B^+$  树的非终端结点。因此，VSAM 文件既可在顺序集中进行顺序存取，又可从最高层的索引（ $B^+$  树的根结点）出发进行按关键字存取。

在 VSAM 文件中，记录可是不定长的，则在控制区间中除了存放记录本身以外，还有每个记录的控制信息和整个区间的控制信息。控制区间的结构如图 10.7 所示。在控制区间上存取一个记录时，需从控制区间的两端出发同时向中间扫描。

### 2. VSAM操作

#### (1) 插入操作

VSAM 文件中没有设置溢出区，解决插入的办法是在初建文件时留有空间：一种方法



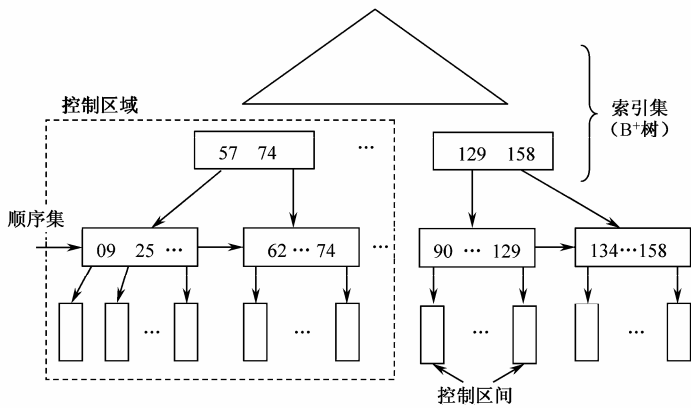


图 10.6 VSAM 文件的结构示意图

记录1	...	记录n	未利用的空闲空间	记录n的控制信息	...	记录1的控制信息	控制区间的控制信息
-----	-----	-----	----------	----------	-----	----------	-----------

图 10.7 控制区间的结构示意图

是每个控制区间内不填满记录，而且在最末一个记录和控制信息之间留有孔隙；另一种方法是在每个控制区域中有一些完全空的控制区间，并在顺序集的索引中指明这些空区间。当插入记录时，大多数新记录都能插入到相应的控制区间内。但要注意，为了保持区间内记录的关键字由小到大有序，需将区间内关键字大于插入记录关键字的记录向控制信息的方向移动。若在若干记录插入之后控制区间已满，则在下一个记录插入时要进行控制区间的分裂，即将近乎一半的记录移到同一控制区域中全空的控制区间中，并修改顺序集中相应的索引项。倘若控制区域中已经没有全空的控制区间，则要进行控制区域的分裂，此时顺序集中的结点亦要分裂，由此还需要修改索引集中的结点信息。但是，由于控制区域较大，一般很少发生分裂的情况。

### (2) 删除操作

在 VASM 文件中删除记录时，需将同一控制区间中比待删除记录关键字大的记录向前移动，把空间留给以后插入的新记录。若整个控制区间变空，则需修改顺序集中相应的索引项。

### 3. VASM文件的优点

和 ISAM 文件相比，基于 B<sup>+</sup>树的 VSAM 文件有如下优点：动态地分配和释放存储空间，可保持平均 75%的存储利用率；能保持较高的查找效率，查找一个后插入记录和查找一个原有记录具有相同的速度；不需要对文件进行重组。因此，基于 B<sup>+</sup>树的 VSAM 文件，通常被作为大型索引顺序文件的标准组织。为了优化性能，VSAM 文件还可采用其他相关技术，如对指针、关键字进行压缩，对索引进行存放处理等。

## 10.6 散列文件

散列文件（也叫直接存取文件）是指利用散列存储方式进行组织的文件。即根据文件中关键字的特点，设计一个散列函数和冲突解决方法，将记录散列到存储设备上。散列文件

中的存储单位称为桶（bucket），桶既可以磁道为单位，也可以块为单位。通常，一个桶能存放  $m$  个记录，一个文件空间分为  $r$  个桶，编号为  $0, 1, 2, \dots, r-1$ 。散列函数  $h(k)$ 就是把关键字的值映射为桶地址。当一个能存放  $m$  个记录的桶中已有  $m$  个记录时，如果再有关键字值映射为该桶地址，则会发生“溢出”，需要将溢出的同义词存放到另一个称为“溢出桶”的桶中，而存放前  $m$  个同义词的桶称为“基桶”。

注意：①溢出桶和基桶的大小相同，相互之间用指针相链接。②当在基桶中没有找到待查记录时，就沿着指针到所指的溢出桶中进行查找，因此同一散列地址的溢出桶和基桶在磁盘上的物理位置不要相距太远，最好在同一柱面上。

假设某一文件有 16 个记录，其关键字分别为：138, 214, 509, 182, 363, 727, 885, 491, 276, 620, 793, 585, 398, 220, 622, 345。桶的容量为  $m=3$ ，桶数  $b=5$ 。用除留余数法作哈希函数  $h(\text{key}) = \text{key} \bmod 5$ 。由此得到的直接存取文件如图 10.8 所示。

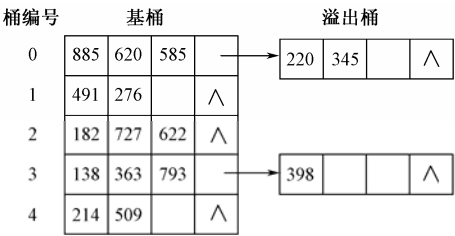


图 10.8 直接存取文件示例

按照散列方法存储记录后，对记录的查找过程为：首先根据给定值求出散列桶地址；然后将基桶中的记录读入内存，进行顺序查找；若找到关键字等于给定值的记录，则检索成功；否则，读入溢出桶中的记录继续进行查找。在散列文件中删去一个记录，仅需对被删记录做删除标记即可。

散列文件的优点是：文件随机存放，记录不需进行排序；插入、删除方便；存取速度快；不需要索引区，节省存储空间。散列文件也有缺点：不能进行顺序存取，只能按关键字随机存取；询问方式限于简单询问；在经过多次插入、删除操作后，可能造成文件结构不合理，需要重新组织文件。

## 10.7 多关键字文件

包含有多个次关键字索引的文件称为多关键字文件，次关键字索引本身可以是顺序表或树表。多关键字文件和其他文件的比较见表 10.1。

表 10.1 多关键字文件和其他文件的比较

	多关键字文件	其 他 文 件
包含的关键字	除主关键字外还有多个次关键字	只含有一个主关键字索引
建立的索引	建立主关键字索引和多个次关键字索引	只有（没有）主关键字索引查询
查询	对主关键字索引或次关键字索引进行查询	只能顺序存取主文件记录进行比较，效率低
文件组织方式	4 种基本组织方法都可以	4 种组织方法都可以

10.7.1 多重表文件

多重表文件（multi-list file）是索引方法和链接方法相结合的一种组织方式，其特点是，记录按主关键字的顺序构成一个串联文件，并建立主关键字的索引（称为主索引）；对每一个次关键字项建立次关键字索引（称为次索引），所有具有同一次关键字的记录构成一个链表。主索引为非稠密索引，次索引为稠密索引。每个索引项包括次关键字、头指针和链表长度。

例如，图 10.9 所示为一个存有某高校教师信息的多重表文件。其中，职工号为主关键字，其他的字段为次关键字，记录按职工号顺序链接。将该表根据不同查找需要分成 3 个子链表，其索引如图 10.9（b）所示，索引项中的主关键字为各子表中的最大值。性别、年龄和所属学院为 3 个次关键字项，它们的索引如图 10.9（c）～（e）所示，具有相同次关键字的记录链接在同一链表中。

记录号	姓名	职工号		性别		年龄		所属学院	
01	李 峰	1200	02	男	03	28	02	电力学院	05
02	罗 娟	1201	03	女	05	25	03	信息学院	06
03	陈明辞	1202	04	男	04	27	07	法学院	10
04	刘伟华	1203	^	男	07	30	06	外语学院	09
05	方 芳	1204	06	女	06	42	10	电力学院	08
06	杨玉琳	1205	07	男	08	37	08	信息学院	07
07	张子翰	1206	08	女	10	29	^	信息学院	^
08	李 莉	1207	^	男	09	31	09	电力学院	^
09	王 萍	1208	10	女	^	39	^	外语学院	^
10	贾 庆	1209	^	男	^	45	^	法学院	^

(a) 数据文件

主关键字	头指针
1203	01
1207	05
1209	09

(b) 主关键字索引

次关键字	头指针	长度
男	01	5
女	02	5

(c) “性别”索引

次关键字	头指针	长度
20~29	01	4
30~39	04	4
40~49	05	2

(d) “年龄”索引

次关键字	头指针	长度
电力学院	01	3
信息学院	02	3
外语学院	04	2
法学院	03	2

(e) “所属学院”索引

图 10.9 多重表文件示例

在多重表文件中进行查询时，若是给定单关键字的简单查询，则根据给定值在对应次关键字索引表中找到对应索引项，从头指针出发，列出该链表上所有记录；若是给定多关键字的关键字组合查询，可先比较两（多）个索引链表的长度，然后选较短的链表进行查找。多重表文件的更新操作包括：① 插入新记录，相同次关键字链表不按主关键字大小链接时，在主文件中插入新记录后，将记录在各个次关键字链表中插到链表的头指针之后即可。② 删除记录，在删去一个记录的同时在每个次关键字的链表中删去该记录。

10.7.2 倒排文件

倒排文件和多重表文件的区别在于次关键字索引结构的不同。在次关键字索引中，具有相同次关键字的记录之间不进行链接，而是列出具有该次关键字记录的物理地址。倒排文

件中的次关键字索引称作倒排表，倒排表和主文件一起构成倒排文件。如图 10.9（a）所示的文件的倒排表如图 10.10 所示。

男	01, 03, 04, 07, 10
女	02, 05, 06, 08, 09

(a) 性别倒排表

20~29	01, 02, 03, 07
30~39	04, 06, 08, 09
40~49	05, 10

(b) 年龄倒排表

电力学院	01, 05, 08
信息学院	02, 06, 07
外语学院	04, 09
法学院	03, 10

(c) 所属学院倒排表

图 10.10 倒排文件索引示例

倒排表的优点是检索记录较快，特别是在处理复杂的多关键字查询时，可在倒排表中先完成查询条件的交、并等逻辑运算，得到结果后再对记录进行存取，即把对记录的查询转换为地址集合的运算来减少对记录的随机存取，从而提高查找速度。倒排文件与一般文件组织的区别在于：在一般的文件组织中，是先找记录，然后再找到该记录所含的各次关键字；而倒排文件是先给定次关键字，然后查找含有该次关键字的各个记录，这种文件查找次序与一般文件的查找次序正好相反，因此称之为“倒排”。值得注意的是，多重表文件实际上也是倒排文件，只不过索引的方法不同。

## 10.8 外部排序

当待排序的对象数目特别多时，在内存中不能一次处理，必须把它们以文件的形式存放于外存中，排序时再把它们一部分一部分地调入内存进行处理。这样，在排序过程中必须不断地在内存与外存之间传送数据。这种基于外部存储设备（或文件）的排序技术就是外部排序。外部排序的过程包括以下步骤：① 按可用内存的大小，将外存中含  $n$  个记录的文件分成若干长度为 1 的子文件或段（segment），依次读入内存并利用有效的内部排序方法对它们进行排序，并将排序后得到的有序子文件〔通常称这些有序子文件为归并段或顺串（run）〕重新写入外存。② 对归并段进行逐趟归并，使归并段（有序的子文件）逐渐由小至大排列，直到整个文件有序为止。

例如，假设有一个含 10000 个记录的文件，首先通过 10 次内部排序得到 10 个初始归并段 R1~R10，其中每一段都含 1000 个记录。然后对它们做如图 10.11 所示的两两归并，直至得到一个有序文件为止。

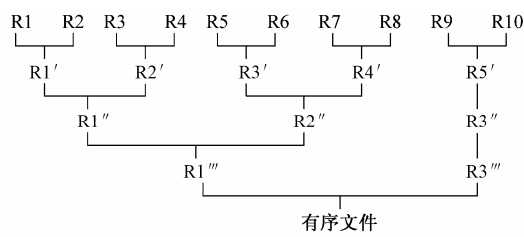


图 10.11 两两归并

由图 10.11 可见, 由 10 个初始归并段到一个有序文件, 共进行了 4 趟归并, 每一趟从  $m$  个归并段到  $\lceil m/2 \rceil$  个归并段。这种归并方法称为 2 路平衡归并。

外部排序所需的总时间 = 内部排序 (产生初始归并段) 所需的时间 ( $mt_{IS}$ ) +  
外存信息读写时间 ( $dt_{IO}$ ) + 内部归并所需的时间 ( $sut_{mg}$ )

其中,  $t_{IS}$  为得到一个初始归并段进行内部排序所需时间的均值;  $t_{IO}$  为进行一次外存读/写时间的均值;  $ut_{mg}$  为对  $u$  个记录进行内部归并所需的时间;  $m$  为经过内部排序之后得到的初始归并段的个数;  $s$  为归并的趟数;  $d$  为总的读/写次数。因此, 上例 1000 个记录利用 2 路归并并进行外排所需总的时间为

$$10 \times t_{IS} + 500 t_{IO} + 4 \times 10000 t_{mg}$$

若对上例中所得的 10 个初始归并段进行 5 路平衡归并 (即每一趟将 5 个或 5 个以下的有序子文件归并成一个有序子文件), 则由图 10.12 可见, 仅需进行两趟归并, 外排的总的读/写次数便减至  $2 \times 100 + 100 = 300$ , 比 2 路归并减少了 200 次的读/写。

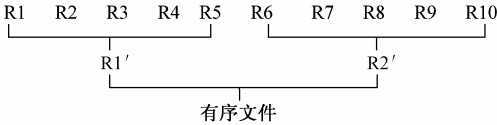


图 10.12 5 路平衡归并

可见, 对同一文件而言, 进行外排时所需读/写外存的次数与归并的趟数  $s$  成正比。在一般情况下, 对  $m$  个初始归并段进行  $k$  路平衡归并时, 归并的趟数  $s = \lfloor \log_k m \rfloor$ 。

# 本章总结

## 1. 学习要点

本章主要介绍了各类文件的基本概念, 文件中数据的数据结构、存储结构、插入/删除运算及实现方法, 以及外部排序的方法, 主要学习要点如下:

- ① 文件的基本概念和特点;
- ② 顺序文件、索引文件、索引顺序文件、散列文件、直接存取文件、多重表文件、倒排文件数据的组织方法和操作特点;
- ③ 如何实现上述文件的检索、插入、删除等操作;
- ④ 外部排序的基本概念和基本方法。

## 2. 基本要求

(1) 清楚文件的概念, 逻辑结构及其运算

- ① 熟悉文件的定义、术语、逻辑结构及其检索和修改两种运算;
- ② 知道文件存储结构的含义、种类以及需要时应考虑的主要因素;
- ③ 清楚外部存储器的有关知识及特点, 分页存储方法的基本概念和术语;

(2) 掌握各种类型文件的组织方法和操作方法

① 掌握顺序文件的组织方法, 在文件上进行操作的方法和其特点以及顺序文件的适用场合。

- ② 清楚索引的基本概念，检索和修改操作方法。
  - ③ 熟悉 ISAM 文件和 VSAM 文件的组织方法和操作方法。
  - ④ 清楚直接存取文件（散列文件）的组织方法，操作方法，查找方法和优缺点。
  - ⑤ 对多关键字文件（多重表文件、倒排文件）有一定程度的了解。
- (3) 掌握外部排序的基本思想和方法
- 掌握归并排序的指导思想和将两个有序文件合并成一个有序文件的算法。

### 3. 重点与难点

本章的重点是：顺序文件、索引文件和散列文件的组织方法和操作方法。难点是：运用顺序文件和索引文件及散列文件的方法编制综合应用程序。

## 习题 10

- 10-1 什么是文件的逻辑记录和物理记录？它们有什么区别与联系？
- 10-2 简述磁带和磁盘的结构和存储信息的特点。
- 10-3 (1) 简述 ISAM 文件组织和操作特点。(2) 简述散列文件的查找方法及优缺点。
- 10-4 文件的组织方法与外存储器的物理特点之间有何联系？试举例说明。
- 10-5 (1) 文件的操作与对内存中数据的操作有何不同？试举例说明。(2) 文件的检索效率取决于哪些因素？
- 10-6 叙述在如图 10.13 所示的 B<sup>+</sup>树上查键值为 85 的记录的过程。

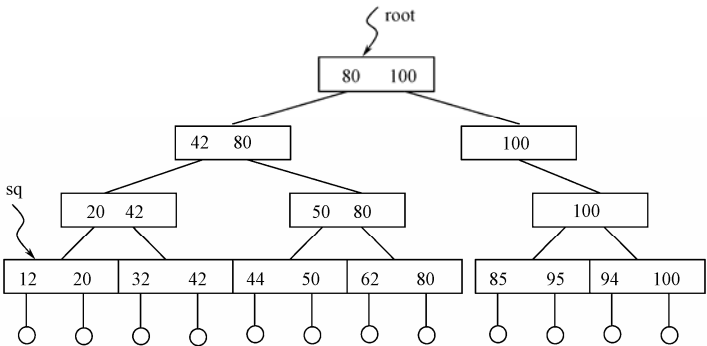


图 10.13 习题 10-6 的图

- 10-7 在如图 10.14 所示的散列文件中，若还有两个键值分别为 132，370 的记录，它们将如何存放？

桶编号	基桶		溢出桶
0	070		NULL
1	512	015	337
2	597	117	NULL
3	262	157	NULL
4	116	613	NULL
5	285	208	817
6	923	076	NULL

图 10.14 习题 10-7 的图

- 10-8 (1) 如何选择合适的文件组织形式? 取决于哪些因素? (2) 简述文件组织的基本方法。
- 10-9 详细说明在索引顺序 (ISAM) 文件中, 查找、增添、删除是如何进行的?
- 10-10 比较总结各种文件组织的建立方式、操作方法、优缺点。
- 10-11 详述对 B<sup>+</sup> 树进行查找、增添、删除操作的方法。B<sup>+</sup> 树的相应操作与 B 树的有何异同?
- 10-12 假设在一个按桶散列的文件组织中, 有  $B$  个桶, 现在要对该文件进行重组织, 准备采用成倍扩大存储桶数目的办法, 把桶扩充到  $2B$  个。请编写算法实现这种重组。

10-13 用直接地址存储方式建立某学院图书管理系统文件, 每个记录包含 5 个域: 书号、书名、类别、作者和出版社, 规定书号按 xxxx/xxx 格式输入, 图书类别分为: 文学、政治、科技、专业、外语、文艺 6 种。本系统要对以下命令综合做出响应:

- add——登录入库的图书 (包括新买进的和读者还书);
- out——输出馆内现有图书的最新清单;
- find——根据书名和作者或只根据书名查书号;
- number——统计馆内各类图书的册数及占总册数的百分比。

10-14 将习题 10-15 改用顺序存储方式, 完成上述 4 个命令的操作。

10-15 设有一个汽车记录文件, 每个记录有 4 个域, 其中汽车号为主关键字, 其他域为次关键字。该文件由 7 个记录组成, 记录形式与每个记录的数据如下:

记录地址	汽车号	汽车号	颜色	产地
101	100	Shanghai	黑色	上海
102	270	Beijing	深色	北京
103	390	Tianjin	红色	天津
104	500	Guangzhou	白色	广州
105	750	Changsha	黑色	长沙
106	950	Shanghai	绿色	上海
107	980	Tianjin	绿色	天津

试用下列结构组织这个文件, 画出这些文件的结构图: (1) 主关键字索引顺序结构; (2) 多重链表结构; (3) 倒排队文件结构。

10-16 设有一个学生记录文件, 每个记录有 5 个域, 其中学号为主关键字, 其他域为辅关键字。该文件由 8 个记录组成, 数据文件如下:

学号	姓名	系别	录取成绩	籍贯	学号	姓名	系别	录取成绩	籍贯
A 850003	张冠	会计	529	上海	E 840242	王文	会计	504	上海
B 870010	李戴	工经	510	山东	F 870100	张伍	金融	497	上海
C 840111	赵明	经济	503	北京	G 850084	刘刚	经贸	578	广东
D 860903	刘七	工经	512	天津	H 870501	董洁	统计	530	江苏

试用下列结构组织这个文件: (1) 主关键字索引顺序结构; (2) 按姓名的字典 (拼音字母) 顺序的索引结构; (3) 散列结构; (4) 链接式结构; (5) 多重表结构; (6) 倒排文件结构。

10-17 写出一个把记录  $R$  插入到具有  $n$  个关键字的多重表文件的算法。假设各个链表中记录是无序的, 算法中可调用  $\text{search}(x)$  和  $\text{update}(x, a)$  来查找和修改索引。 $\text{update}(x, a)$  的作用是把  $x$  的地址改为  $a$ 。试问: 算法共需多少次访问磁盘 (为修改索引所需的次数除外)?

10-18 请写出一个可从多重表文件中删除任意记录  $R$  的算法。该算法需要多少次访问磁盘?

- 10-19 假定各个链表是按主关键字  $R \rightarrow \text{link}$  递增排序的，请写出一个可将记录  $R$  插入多重表文件的算法，并分析算法的效率（不计索引存取）。
- 10-20 假定多重表文件中的每一个链表都是双向链表，请写出一个可删除任意记录  $R$  的算法。对应于记录  $R$  的第  $i$  个关键字的直接前驱指针为  $R \rightarrow \text{ialink}$ ，而相应的直接后继指针为  $R \rightarrow \text{iblink}$ 。
- 10-21 编写可从一个循环链表文件中删除任意记录  $R$  的算法。
- 10-22 编写能对习题 10-16 中的学生记录文件建立多重表文件的算法。
- 10-23 假设某大学某系有一份职工文件，文件中的每一个记录都包含下列诸域：

职 工 号	姓 名	性 别	民 族	年 龄	职 称	职 务	本 人 成 分	工 次	政 治 面 貌	文 化 程 度	健 康 状 况	会 几 门 外 语
-------------	--------	--------	--------	--------	--------	--------	------------------	--------	------------------	------------------	------------------	-----------------------

- 设“职工号”域为主关键，其他域为次关键字，试用下列结构组织这份职工文件：（1）索引顺序结构；（2） $B^+$ 树索引结构；（3）倒排文件结构。选择合适的计算机程序语言，编出查找和显示文件中任何一部分或修改文件等功能的完整程序，并在计算机上调试通过。
- 10-24 编写从一个多表文件删除一个任意的记录  $z$  的算法（参考习题 10-25），需要多少次磁盘存取？
- 10-25 假定各表都按主关键字  $\text{key}(z, 1)$  排序，编写将记录  $z$  插入到一个多表文件中的算法，需要多少次磁盘存取（除去索引的存取）？



# 第 11 章 贪婪算法

前面的章节介绍了数据结构及其存储实现方式和相关操作算法,从这一章开始,将讨论算法设计问题。由于目前尚不存在能够用来设计所有算法的技术,因此一个好的算法设计方法更像是一门艺术。尽管如此,仍然存在一些行之有效和具有普适性的算法设计方法。本书第 11~15 章提供了贪婪算法、分而治之算法、动态规划、回溯和分枝定界法 5 种基本的算法设计方法。对于许多问题,这些方法中至少有一种是可以解决的。下面主要通过实例来介绍这些方法的基本思想及其工作过程。本章介绍一种直观的问题求解方法——贪婪算法。本章先从最优化概念开始,然后介绍该算法在货箱装船问题、背包问题、拓扑排序问题、二分覆盖问题、最短路径问题、最小代价生成树问题中应用时的求解方案。

## 11.1 最优化问题

从这一章开始所列举的实例大多属于最优化问题。一个最优化问题通常包含一个基本问题、一组限制条件和一个优化函数。满足限制条件的问题求解方案称为可行解,所有可行解组成的集合为该问题的解空间,使优化函数取得最佳值的可行解称为最优解。下面通过一个例子来说明最优化问题。

**【例 11-1】** 有一个非常聪明的小孩,当他很渴时可获得的饮品包括水、牛奶、多种不同种类的果汁和苏打水。他通过对各种饮料饮用 20 毫升的方法来为每一种饮料给予一个满意度值,即对第  $i$  种饮料,其满意度为  $s_i$ 。通常,这个小孩会选择具有最大满意度值的饮料来满足解渴的需要,设  $a_i$  是第  $i$  种饮料的总量(以毫升为单位),小孩总共需要  $t$  毫升饮料才能完全解渴。如果具有最大满意度值的饮料没有足够的量来满足其解渴需求,那么,小孩需要饮用  $n$  种不同的饮料各多少才能满足他的解渴需求呢?

设各种饮料的满意度已知。令  $x_i$  为小孩将要饮用的第  $i$  种饮料的量,则此问题的解决可描述为找到一组实数  $x_i$  ( $1 \leq i \leq n$ ),使  $\sum_{i=1}^n s_i x_i$  最大,并满足  $\sum_{i=1}^n x_i = t$  及  $0 \leq x_i \leq a_i$ 。对上述问题的输入/输出用数学公式进行形式化描述如下。

输入:  $n, t, s_i, a_i$  (其中  $1 \leq i \leq n, n$  为整数,  $t, s_i, a_i$  为正实数)。

输出: 实数  $x_i$  ( $1 \leq i \leq n$ ),使  $\sum_{i=1}^n s_i x_i$  最大且  $\sum_{i=1}^n x_i = t$ 。如果  $\sum_{i=1}^n a_i < t$ ,则输出适当的错误信息。

在这个问题中,限制条件是  $\sum_{i=1}^n x_i = t$  且  $0 \leq x_i \leq a_i, 1 \leq i \leq n$ ,而优化函数是  $\sum_{i=1}^n s_i x_i$ 。任何满足限制条件的一组实数  $x_i$  都是可行解,而使  $\sum_{i=1}^n s_i x_i$  最大的可行解是最优解。

## 11.2 算 法 思 想

贪婪算法是采用逐步构造最优解的问题解决方法，即在每个阶段都做出在当前状态下看上去最优的选择，并希望通过每次的贪心选择而使最终结果是问题的最优解。其中，做出贪婪选择的依据称为贪婪准则。贪心算法并不能保证一定得到最优解，但在很多情况下确实能达到预期的效果或者与最优解很接近。

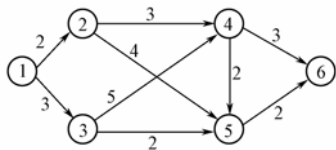


图 11.1 有向图

**【例 11-2】** 给出一个有向网络，如图 11.1 所示。路径的长度定义为路径所经过的各边的耗费之和。要求找一条从初始顶点 1 到达目的顶点 6 的最短路径。

该问题利用贪婪算法来分步构造这条路径，每一步在路径中加入一个顶点，假设当前路径已到达顶点  $p$ ，且顶点  $p$  并不是目的顶点  $r$ ，则下一个顶点的加入可采用的贪婪准则为：选择离  $p$  最近且目前不在路径中的顶点。利用上述贪婪算法，从顶点 1 开始并寻找目前不在路径中离顶点 1 最近的顶点为 2，从顶点 2 可到达的最近顶点为 4，从顶点 4 到达顶点 5，最后到达目的顶点 6。所建立的路径为 1, 2, 4, 5, 6，其长度为 9。但这种贪婪算法并不一定能获得最短路径，事实上，还存在其他更短的路径，如路径 1, 3, 5, 6，其长度为 7。

虽然贪婪算法在解决一些问题时并不能保证得到最优解，但通常所得的结果与最优解相差无几，因此，这种算法也称为启发式方法，是一种近似算法。对于很多用贪心算法来求解的问题，它们一般都具有两个重要的性质（贪心选择性质和最优子结构性质）。所谓贪心选择性质，是指问题的整体最优解是通过一系列贪心选择来获得的，每一步的贪心选择都是在当前状况下看起来最好的选择，即局部最优解，然后以此为基础再进行后续的贪心选择。由此可见，贪心算法是一种从上到下的迭代选择方法，每一步贪心选择都使得所求解问题的规模变小。而最优子结构性质是指贪心算法所得到某问题的最优解包含了其子问题的最优解。下面将介绍贪婪算法的几种应用，在这些应用中，贪婪算法有时能得到最优解，有时只是一种启发式方法，可能获得的不是最优解。

## 11.3 应 用

### 11.3.1 货箱装船

有一艘准备用来装载货物的大船，所有待装货物都装在大小一样、重量不同的货箱中。设第  $i$  个货箱的重量为  $w_i$  ( $1 \leq i \leq n$ )，而货船的最大载重量为  $c$ ，问如何在货船上装入最多的货物。

对这个问题的求解可按照贪婪算法进行分步装载，采用的贪婪准则为：每次装载时，从剩下的货箱中选择重量最小的货箱。根据此贪婪策略，首先选择最轻的货箱，然后选择较轻的货箱，如此下去，直到所有货箱均装上船或船上的货物重量超过所能承受的最大重量。这种选择次序可保证所选的货箱总重量最小，而装载的货箱数量最多。

货箱装载算法的 C++代码见程序 11-1。由于贪婪算法是按货箱重量递增的顺序装载的，所以程序 11-1 首先利用间接寻址排序函数 TableInsertSort 对货箱重量进行排序，随后货箱便可按重量递增的顺序装载。由于间接寻址排序所需的时间为  $O(n \lg n)$ ，该算法其余部分所需时间为  $O(n)$ ，因此程序 11-1 总的复杂性为  $O(n \lg n)$ 。

程序 11-1 货箱装载算法

```
template<class T>
void ContainerLoading(int x[ ], SLinkListType<T> &SL, T c)
{ //货箱装船问题的贪婪算法
  //x[i]=1 当且仅当货箱 i 被装载，1≤i≤n
  //c 是船的容量，SL 中的关键字用来记录货箱的重量
  //对重量按间接寻址方式排序
  TableInsertSort(SL); //使用程序 9-5 中的算法
  //初始化 x
  for (int i=1; i<=SL.length; i++) x[i]=0; //按重量次序选择物品
      int j=SL.r[0].next;
      for (i=0; i<=SL.length && SL.r[j].rc.key<=c; i++)
      {   x[j]=1; c-=SL.r[j].rc.key; //剩余容量
          j=SL.r[j].next;
      }
}
```

11.3.2 0-1 背包问题

0-1 背包问题是：从  $n$  个重量分别为  $w_i$ 、价值分别为  $p_i$  的物品中选取部分物品装入总容量为  $c$  的背包中，使背包中物品总重量不超过背包的总容量且所物品的总价值最高，即在满足  $\sum_{i=1}^n w_i x_i \leq c$  且  $x_i \in [0, 1]$  ( $1 \leq i \leq n$ ) 的条件下使  $\sum_{i=1}^n p_i x_i$  最大。假设用  $x_i = 1$  表示物品  $i$  装入背包中， $x_i = 0$  表示物品  $i$  不装入背包，因此该问题需要求出  $x_i$  的值，即各物品装入与否的情况。0-1 背包问题可有几种贪婪策略。

第一种为价值贪婪准则，即每次都从剩余物品中选择价值最大的物品装入背包。在此规则下，物品按照其价值由大到小依次装入背包，直到物品重量超过背包的最大容量。这种策略不能保证得到最优解。例如， $n = 2$ ， $w = [100, 10, 10]$ ， $p = [20, 15, 15]$ ， $c = 105$ 。当利用价值贪婪准则时，获得的解为  $x = [1, 0, 0]$ ，这种方案的总价值为 20。而最优解为  $[0, 1, 1]$ ，其总价值为 30。

第二种为重量贪婪准则，即每次都从剩余物品中选择重量最小的物品装入背包。这种策略虽然对前面的例子可产生最优解，但在一般情况下不一定能得到最优解。例如， $n = 2$ ， $w = [10, 20]$ ， $p = [5, 100]$ ， $c = 25$ 。当利用重量贪婪准则时，获得的解为  $x = [1, 0]$ ，比最优解  $[0, 1]$  要差。

第三种为价值密度  $p_i/w_i$  准则，即每次从剩余物品中选择  $p_i/w_i$  值最大的物品装入背包。这种策略也不能保证得到最优解。

由此可见，0-1 背包问题是一个 NP-复杂问题，NP-复杂问题及 NP-完全问题是指尚未找

到具有多项式时间复杂度算法的问题。NP-完全问题是一类判断问题，即这类问题的答案为是或否。NP-复杂问题可能是判断问题，也可能不是判断问题。现实世界中，成千上万的具有实际意义的问题都是 NP-复杂问题或 NP-完全问题，如果有人能对一个 NP-复杂问题或 NP-完全问题找到一个多项式算法，那么他同时也找到了能在多项时间内解决所有 NP-复杂问题或 NP-完全问题的方法。虽然不能证明 NP-完全问题不能在多项式时间内得到解决，但大家都认为这已是一个事实。因此，NP-复杂问题的优化问题经常用近似算法解决，虽然近似算法不能保证得到最优解，但能保证所获得的是近似最优解。

### 11.3.3 拓扑排序

一个复杂工程通常可分解成多个子任务，子任务之间具有先后关系，这种先后顺序可用有向图（AOV）来表示，其中顶点代表各个任务，有向边  $(i, j)$  表示任务之间的先后关系，即在任务  $j$  开始前，任务  $i$  必须完成，这种序列也称为拓扑序列。根据任务的有向图建立拓扑序列的过程称为拓扑排序。

**【例 11-3】** 如图 11.2 所示为包含 6 个任务的工程的有向图，图中有多种拓扑序列，如 123456、132456、215346 等，但序列 142356 不是拓扑序列，因为在这个序列中任务 4 在 3 的前面，而任务有向图中的边为  $(3, 4)$ ，这种序列与边  $(3, 4)$  及其他边所指示的序列相矛盾。

任务有向图的拓扑序列可用贪婪算法来求得，序列中每一个顶点的选择可按照如下贪婪准则进行：从剩下的顶点中选择顶点  $w$ ，使得  $w$  不存在这样的入边  $(v, w)$ ，其中顶点  $v$  不在已排好的序列结构中出现。算法按从左到右的步骤构造拓扑序列，每一步在排好的序列中加入一个新顶点。

以图 11.2 为例，贪婪算法的求解过程如下：①从一个空序列  $V$  开始，选择  $V$  的第一个顶点，此时有两个候选顶点 1 和 2，若选择顶点 1，则序列  $V=1$ ；②选择  $V$  的第二个顶点，根据贪婪准则可知候选顶点为 2 和 3，若选择 3，则  $V=13$ ；③顶点 2 是唯一的候选，因此  $V=132$ ；④候选顶点为 4 和 5，若选择 4，则  $V=1324$ ；⑤分别加入顶点 5 和 6，得  $V=132456$ 。贪婪算法的伪代码如图 11.3 所示。

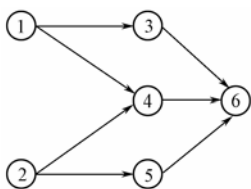


图 11.2 任务有向图

```

设  $n$  是有向图中的顶点数， $I$  是一个空序列
while (true)
{ 设  $w$  不存在入边  $(v, w)$ ，其中顶点  $v$  不在  $I$  中
  如果没有这样的  $w$ ，则 break
  把  $w$  添加到  $I$  的尾部
}
if ( $I$  中的顶点数小于  $n$ ) 算法失败

```

图 11.3 拓扑排序

### 11.3.4 二分覆盖

二分图是一个无向图，它的  $n$  个顶点分为两个集合  $A$  和  $B$ ，图中任何一条边的两个顶点分别属于两个不同的集合，即同一集合中的任意两个顶点在图中无边相连。当且仅当  $B$  中的每个顶点至少与  $A$  中一个顶点相连时， $A$  的一个子集  $A'$  覆盖集合  $B$ （或者简单地说， $A'$  是一个覆盖）。覆盖  $A'$  的大小即为  $A'$  中的顶点数目。当且仅当  $A'$  是覆盖  $B$  的子集中最小子集时， $A'$  为最小覆盖。在如图 11.4 所示的包含 17 个顶点的二分图中， $A=\{1, 2, 3, 16, 17\}$  和  $B=\{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$ ，子集  $A'=\{1, 16, 17\}$  是  $B$  的最小覆盖。

二分覆盖问题是指在二分图中寻找最小覆盖的问题，这个问题类似于集合覆盖问题：

给出  $k$  个集合  $S = \{S_1, S_2, \dots, S_k\}$ , 每个集合  $S_i$  中的元素均是全集  $U$  中的成员。当且仅当  $\bigcup_{i \in S'} S_i = U$  时,  $S$  的子集  $S'$  覆盖  $U$ ,  $S'$  中的集合数目即为覆盖的大小。当且仅当没有能覆盖  $U$  的更小的集合时, 称  $S'$  为最小覆盖。由此可见, 集合覆盖问题和二分覆盖问题可相互转化, 即用  $A$  的顶点来表示  $S_1, S_2, \dots, S_k$ ,  $B$  中的顶点代表  $U$  中的元素, 当且仅当  $S$  的相应集合中包含  $U$  中的对应元素时, 在  $A$  与  $B$  的顶点之间存在一条边。

**【例 11-4】** 令  $S = \{S_1, S_2, \dots, S_5\}$ ,  $U = \{4, 5, \dots, 15\}$ ,  $S_1 = \{4, 6, 7, 8, 9, 13\}$ ,  $S_2 = \{4, 5, 6, 8\}$ ,  $S_3 = \{8, 10, 12, 14, 15\}$ ,  $S_4 = \{5, 6, 8, 12, 14, 15\}$ ,  $S_5 = \{4, 9, 10, 11\}$ 。 $S' = \{S_1, S_4, S_5\}$  是一个大小为 3 的覆盖, 没有更小的覆盖,  $S'$  即为最小覆盖。这个集合覆盖问题可映射为图 11.4 所示的二分图, 即用顶点 1, 2, 3, 16 和 17 分别表示集合  $S_1, S_2, S_3, S_4$  和  $S_5$ , 顶点  $j$  表示集合中的元素  $j$ ,  $4 \leq j \leq 15$ 。

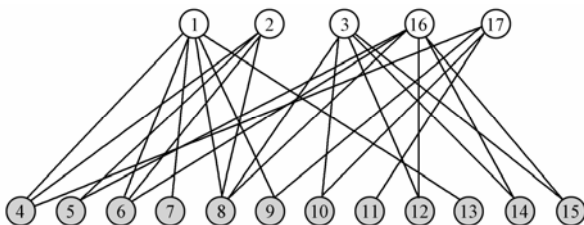


图 11.4 二分图示例

二分覆盖与集合覆盖均为 NP—复杂问题, 因此可能无法找到一个快速的算法来实现, 但贪婪算法可作为解决该问题的一种快速启发式方法。定点的选择根据如下贪婪准则: 每次从  $A$  中选取能覆盖  $B$  中还未被覆盖的元素数目最多的顶点, 通过这种策略每次选择  $A$  中的一个顶点加入覆盖, 直到建立起覆盖  $A'$ 。

程序 11-2 给出二分覆盖类 `BMCover` 定义, 程序 11-3 给出所使用的无向图的类定义, 程序 11-4 给出所使用无向图的类的成点函数, 程序 11-5 给出了 `BMCover` 的成员函数实现。其中, 成员 `bg` 是所对应的集合, 用无向图来记录。(本书第 7 章讨论了有向图的类及实现。)

程序 11-2 二分覆盖类 `BMCover` 定义

```
template <class VertexType, class EdgeType>
class BMCover
{
    void CreateBins(int b, int n); //创建 b 个空箱子和 n 个结点
    void DestroyBins() { if (node)delete [ ] node; if (bin)delete [ ] bin; }
    void InsertBins(int b, int v); //在箱子 b 中添加顶点 v
    void MoveBins(int bMax, int ToBin, int v);
    //从当前箱子中移动顶点 v 到箱子 ToBin 中
    int *bin; //bin[i]指向代表该箱子的双向链表的首结点
    NodeType *node; //node[i]代表存储顶点 i 的结点
public:
    GraphALND<VertexType, EdgeType> bg; //集合所对应的无向图
    BMCover(long vnum): bg(vnum) { };
    virtual ~BMCover() { };
    bool BipartiteCover(int C[ ], int &m);
};
```

### 程序 11-3 所使用的无向图的类定义

```
template <class VertexType, class EdgeType>
class GraphALND:public GraphAOV<VertexType, EdgeType>
{
//在无向图中, 从 GraphAOV 中所继承的 InDegree 数组只是表示度
public:
    GraphALND(long num);
    virtual ~GraphALND( ) {} ;
    virtual bool AddEdge(long v1, long v2, EdgeType &info, float w);
};
```

### 程序 11-4 所使用的无向图的类的成员函数

```
template <class VertexType, class EdgeType>
bool GraphALND<VertexType, EdgeType>::AddEdge(long v1, long v2, EdgeType &info, float w)
{
//在 v1, v2 之间添加一条边
    return SetEdge(v1, v2, info, w)&& SetEdge(v2, v1, info, w);
}

template<class VertexType, class EdgeType>
GraphALND<VertexType, EdgeType>::GraphALND(long num)
:GraphAOV<VertexType, EdgeType>(num) {} ;
```

### 程序 11-5 BMCover 的成员函数

```
template <class VertexType, class EdgeType>
void BMCover<VertexType, EdgeType>::CreateBins(int b, int n)
{
    node=new NodeType [n+1]; bin=new int [b+1]; //创建 b 个空箱子和 n 个结点
    for (int i=1; i<=b; i++) bin[i]=0; //将箱子置空
}

template<class VertexType, class EdgeType>
void BMCover<VertexType, EdgeType>::InsertBins(int b, int v)
{
//若 b 不为 0, 则将 v 插入箱子 b 中
    if (!b) return; //b 为 0, 不插入
    node[v].left=b; //添加在左端
    if (bin[b]) node[bin[b]].left=v; node[v].right=bin[b]; bin[b]=v;
}

template<class VertexType, class EdgeType>
void BMCover<VertexType, EdgeType>::MoveBins(int bMax, int ToBin, int v)
{
//将顶点 v 从其当前所在箱子移动到 ToBin 中
    int l=node[v].left; int r=node[v].right; //v 的左、右结点
    //从当前箱子中删除
    if (r) node[r].left=node[v].left;
    if (l>bMax || bin[l]!=v) node[l].right=r; //不是最左结点
    else bin[l]=r; //箱子 l 的最左边
```

```

        //添加到箱子 ToBin
        InsertBins (ToBin, v);
    }

```

程序 11-6 为贪婪覆盖算法的完整代码实现。

### 程序 11-6 贪婪覆盖算法

```

template <class VertexType, class EdgeType>
bool BMCover<VertexType, EdgeType>::BipartiteCover(int C[ ], int &m)
{
    //寻找一个二分图覆盖, L 是输入顶点的标号, L[i]=1 当且仅当 i 在 A 中
    //C 是一个记录覆盖的输出数组
    //如果图中不存在覆盖, 则返回 false, 如果图中有一个覆盖, 则返回 true
    //在 m 中返回覆盖的大小, 在 C[0:m-1]中返回覆盖

    //插件结构
    int SizeOfA=0; int n=bg.NumOfVertexes( ); //图中的顶点数
    GraphNode<VertexType> node;
    for (int i=0; i<n; i++)//确定集合 A 的大小
    {
        if (bg.GetVertex(i, node))
            if (node.GetInfo( )==1)SizeOfA++;
    }
    int SizeOfB=n-SizeOfA; CreateBins(SizeOfB, n);
    int*New=new int[n+1]; //顶点 i 覆盖了 B 中 New[i]个未被覆盖的顶点
    bool*Change=new bool[n+1]; //Change[i]为 true 当且仅当 New[i]已改变
    bool*Cov=new bool[n+1]; //Cov[i]为 true 当且仅当顶点 i 被覆盖
    SeqStack<int> S(n);
    //初始化
    for (i=1; i<=n; i++)
    {
        Cov[i]=Change[i]=false;
        if (bg.GetVertex(i-1, node))
            if (node.GetInfo( )==1)
            {
                //i 在 A 中
                New[i]=bg.GetInDegree( )[i-1]; //i 覆盖的
                InsertBins(New[i], i);
            } //if
    } //for
    //构造覆盖
    int covered=0; //被覆盖的顶点
    MaxBin=SizeOfB; //可能非空的箱子
    m=0; //C 的游标
    while (MaxBin>0)
    {
        //搜索所有箱子, 选择一个顶点
        if (bin[MaxBin])
            //箱子不空

```

```

int v=bin[MaxBin]; //第一个顶点
C[m++]=v; //把 v 加入覆盖
//标记新覆盖的顶点
int j=bg.GetFirstNeighbor(v), k; //j 是 v 的第一个邻接的顶点
while (j>=0)
{   if (!Cov[j])
    { //j 尚未被覆盖
        Cov[j]=true; covered++;
        //修改 New
        k=bg.GetFirstNeighbor(j); //k 是 j 的第一个邻接的顶点
        while (k>=0)
        {   New[k]--; //j 不计入在内
            if (!Change[k])
            {   S.Push(k); //仅入栈一次
                Change[k]=true; } //if
            k=bg.GetNextNeighbor(j, k);
            //循环查找 j 的下一个邻接的顶点
        } //while
    } //if
    j=bg.GetNextNeighbor(v, j); //循环查找 v 的下一个邻接的顶点
} //while
//更新箱子
while (!S.IsEmpty( )) { S.Pop(k); Change[k]=false;
    MoveBins(SizeOfB, New[k], k); } //while
MaxBin--; } //while
else MaxBin--; } //for
DestroyBins( ); delete[ ]New; delete[ ]Change; delete[ ]Cov;
return(covered==SizeOfB);
}

```

---

### 11.3.5 单源最短路径

**Dijkstra 算法：**给出一个有向图  $G$ ，假设图中每条边的权值为该边所连接两个顶点的距离，则图中任意两个顶点之间的路径长度为所经过边的权值之和。单源最短路径问题为：对于有向图  $G$  中的源顶点  $a$ ，求出它到图中其他任意顶点的最短路径。

**【例11-5】** 图 11.5 (a) 给出了一个包含 5 个顶点的有向图，各边的权值即为长度。假设源顶点  $a$  为 1，则从顶点 1 出发到图中其他顶点的最短路径（按路径长度顺序）如图 11.5 (b) 所示。

利用 Dijkstra 发明的贪婪算法可求解单源最短路径问题，算法思想是：按路径长度递增的次序产生最短路径，采用的贪婪准则是：在未产生最短路径的顶点中，选择路径长度最短的目的顶点。它通过分步方法求出最短路径，每一步产生一个到达新的目的顶点的最短路径，即它是按照路径长度顺序产生最短路径的。



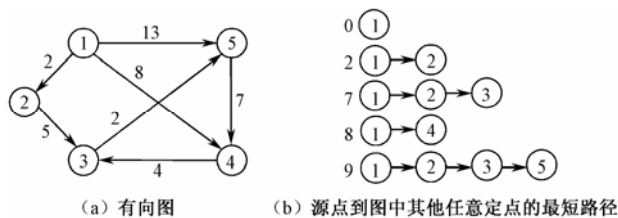


图 11.5 最短路径举例

Dijkstra 算法描述如下。

① 假设用带权的邻接矩阵 `arcs` 来表示带权有向图，`arcs[i][j]` 表示弧  $\langle v_i, v_j \rangle$  上的权值。若  $\langle v_i, v_j \rangle$  不存在，则置 `arcs[i][j]` 为  $\infty$ （在计算机上可用允许的最大值代替）。 $S$  为已找到从  $v$  出发的最短路径的终点集合，它的初始状态为空集。那么从  $v$  出发到图上其余各顶点（终点） $v_i$  可能达到的最短路径长度的初值为  $D[i] = \text{arcs}[\text{LocateVex}(G, v)][i]$ ,  $v_i \in V$ 。

② 选择  $v_j$ ，使得  $D[j] = \text{Min}\{D[i] \mid v_i \in V-S\}$ ， $v_j$  就是当前求得的一条从  $v$  出发的最短路径的终点，令  $S = S \cup \{j\}$ 。

③ 修改从  $v$  出发到集合  $V-S$  上任一顶点  $v_k$  可达的最短路径长度。如果  $D[j] + \text{arcs}[j][k] < D[k]$ ，则修改为  $D[k] = D[j] + \text{arcs}[j][k]$ 。

④ 重复操作②、③共  $n-1$  次，由此求得从  $v$  到图上其余各顶点的最短路径是依路径长度递增的序列。

求有向图的最短路径，数据存储类型实用邻接矩阵，故算法设置为 `GraphMatrix` 类的成员函数。

最短路径算法见程序 11-7。

程序 11-7 最短路径算法

```
template <class VertexType, class EdgeType>
void GraphMatrix<VertexType, EdgeType>::ShortestPath_DIJ(int v0, int **P, float D[ ])
{
    //用 Dijkstra 算法求有向网 G 的 v0 顶点到其余顶点 v 的最短路径 P[v] 及其带权长度 D[v]
    //若 P[v][w] 为 TRUE，则 w 是从 v0 到 v 当前求得最短路径上的顶点
    //final[v] 为 TRUE 当且仅当 v ∈ S，即已经求得从 v0 到 v 的最短路径
    int i, j, k, l; float w, min;
    bool *final=new bool[numNodes]; //最短路径顶点集
    for (i=0; i<numNodes; i++)
    {
        final[i]=false;
        D[i]=edges[v0][i].GetWeight(); //数组初始化
        for (j=0; j<numNodes; j++) P[i][j]=0;
        if (i!=v0 && D[i]<INFINITY) { P[i][v0]=1; P[i][j]=1; }
        else P[i][v0]=-1;
        final[v0]=true; D[v0]=0; //顶点 v0 加入顶点集合
        for (l=0; l<numNodes; l++)
        {
            min=INFINITY; int u=v0; //选不在 S 中具有最短路径的顶点 u
            for (j=0; j<numNodes; j++)
                if (final[j]==false && D[j]<min) { u=j; min=D[j]; }
```

```

        final[u]=true; //将顶点 u 加入集合 S
        for (k=0; k<numNodes; k++) //修改
            //顶点 k 未加入 S, 且绕过 u 可以缩短路径
            if (final[k]==false && w<INFINITY && D[u]+w<D[k])
            {
                w=edges[u][k].GetWeight();
                D[k]=D[u]+w;
                for (int m=0; m<numNodes; m++) P[k][m]=P[u][m];
                P[k][k]=true; //修改到 k 的最短路径
            } //if
    } //for
} //for
} //ShortestPath-DIJ

```

---

### 11.3.6 最小代价生成树

**【例 11-6】** 假设要在  $n$  个城市之间建立通信网络, 而连通  $n$  个城市一般只需要  $n-1$  条线路。由于  $n$  个城市中的每两个城市之间都可建立一条线路, 因此最多可有  $n(n-1)/2$  条线路可供选择, 本问题的关键就是如何在这些可能的  $n(n-1)/2$  条线路中选择  $n-1$  条线路, 使得建立这个通信网的成本最低。

$n$  个城市及其可能设置的通信线路可用连通网来表示, 其中网的顶点表示城市, 边表示两城市之间的线路, 边的权值表示相应的代价。由于具有  $n$  个顶点的连通网包含多棵不同的生成树, 每一棵生成树都可能是一个通信网且总代价不同, 因此该问题的解决就要通过构造连通网的最小代价生成树来完成。

最小代价生成树的构造有多种算法, 其中大多数算法利用了最小生成树的 MST 性质, 该性质描述为: 假设  $N = (V, \{E\})$  是一个连通网,  $U$  是顶点集  $V$  的一个非空子集, 若  $(u, v)$  是一条具有最小权值 (代价) 的边, 其中  $u \in U, v \in V-U$ , 则必存在一棵包含边  $(u, v)$  的最小生成树。

上述性质可用反证法来证明。假设网  $N$  的任何一棵最小生成树都不包含  $(u, v)$ , 设  $T$  是连通网上的一棵最小生成树, 当将边  $(u, v)$  加入到  $T$  中时, 由生成树的定义,  $T$  中必存在一条包含  $(u, v)$  的回路。另外, 由于  $T$  是生成树, 则在  $T$  上必存在另一条边  $(u', v')$ , 其中  $u' \in U, v' \in V-U$ , 且  $u$  和  $u'$  之间,  $v$  和  $v'$  之间均有路径相通, 删去边  $(u', v')$ , 便可消除上述回路, 同时得到另一棵生成树  $T'$ 。因为  $(u, v)$  的代价不高于  $(u', v')$ , 所以  $T'$  的代价亦不高于  $T$ ,  $T'$  是包含  $(u, v)$  的一棵最小生成树。由此与假设矛盾。

下面介绍的普里姆 (Prim) 算法和克鲁斯卡尔 (Kruskal) 算法, 就是两个利用 MST 性质构造最小生成树的算法。

#### 1. 普里姆 (Prim) 算法

假设  $N = (V, \{E\})$  是连通网,  $TE$  是  $N$  上最小生成树中边的集合。算法从  $U = \{u_0\}$  ( $u_0 \in V$ ),  $TE = \{\}$  开始, 重复执行下述操作: 在所有  $u \in U, v \in V-U$  的边  $(u, v) \in E$  中找一条代价最小的边  $(u_0, v_0)$  并入集合  $TE$  中, 同时  $v_0$  并入  $U$  中, 直至  $U = V$  为止。此时,  $TE$  中必有  $n-1$  条边, 则  $T = (V, \{TE\})$  为  $N$  的最小生成树。

mst 数组用于存放最小生成树。最小生成树的具体构造过程如下。

- ① 任取一个顶点 0 加入生成树中，mst 中存放从顶点 0 到其余各顶点的边，若无边，则将权值置为(INFINITY)。
- ② 当前 mst[0]到 mst[n-2]中存放的边，都是一个顶点在生成树中，另一个顶点不在生成树中的边，从中选出权值最小的边 mst[min]，将其加入到最小生成树中，即将 mst[0]与 mst[min]互换。Mst[0].startNo 是新加入到生成树的顶点，设该新加入的顶点为顶点 k。
- ③ 调整 mst[1]到 mst[n-2]。若不在生成树的边集中，从顶点 s 到顶点 k 的边上权值比原来顶点 s 到生成树中顶点的边上权值小，就需要将原来的边调整为(k, s)。
- ④ 从 mst[i]到 mst[n-2]重复上述操作②和③ (i=1, 2, ..., n-3)，每次当一个顶点在生成树中，另一个顶点不在生成树的边中时，选出一个权值最小的边，将这条边和不在生成树中的顶点加入到生成树中，并调整未加入到生成树的那些边，直到 n-1 条边都在生成树中，或者选出一条边上权值最小的边，其权值为 $\infty$ ，则该无向图网络不连通，不能构造一颗最小生成树为止。

例如，构造图的最小生成树的过程如图 11.6 所示。

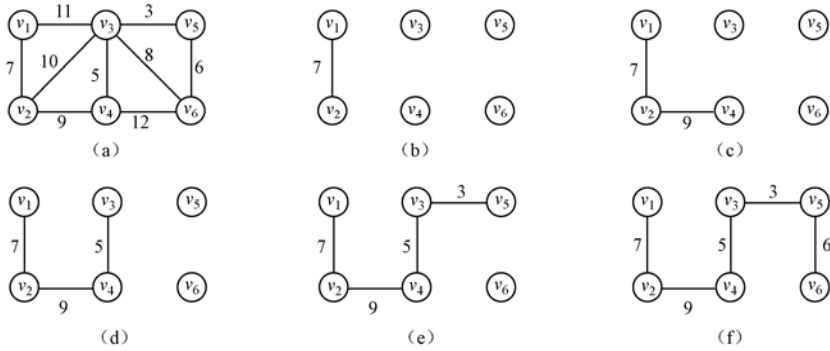


图 11.6 Prim 算法构造最小生成树的过程

程序 11-8~程序 11-10 分别给出了采用数组表示法的无向图 GraphMatrixND 的类及其成员函数的定义以及基于 Prim 算法的最小生成树生成算法。

程序 11-8 邻接矩阵无向图的类描述

```
template <class VertexType, class EdgeType>
class GraphMatrixND : public GraphMatrix<VertexType, EdgeType>{
private:
    void Swap(GraphEdgeMatrix<EdgeType> &to,
              GraphEdgeMatrix<EdgeType> &from);
public:
    GraphMatrixND(int num=MAX_VERTEX_NUM);
    virtual ~GraphMatrixND() {} ;
    virtual bool AddEdge(long v1, long v2, EdgeType &info, float w);
    virtual bool MiniSpanTree_PRIM(GraphEdgeMatrix<EdgeType> *mst);
};
```

```
template <class VertexType, class EdgeType>
bool GraphMatrixND<VertexType, EdgeType>::AddEdge(long v1, long v2, EdgeType
&info, float w) { //在 v1, v2 间添加一条边
    return SetEdge(v1, v2, info, w)&& SetEdge(v2, v1, info, w); }
template<class VertexType, class EdgeType>
GraphMatrixND<VertexType, EdgeType>::GraphMatrixND(int num=MAX_VERTEX_NUM):
GraphMatrix<VertexType, EdgeType>(num) {; }
template<class VertexType, class EdgeType>
void GraphMatrixND<VertexType, EdgeType>::Swap(GraphEdgeMatrix
<EdgeType> &to, GraphEdgeMatrix<EdgeType> &from) { //辅助函数
    GraphEdgeMatrix<EdgeType> temp;
    temp=to; to=from; from=temp;
}
```

### 程序 11-10 最小生成树生成算法

```
template <class VertexType, class EdgeType>
bool GraphMatrixND<VertexType, EdgeType>::
MiniSpanTree_PRIM(GraphEdgeMatrix<EdgeType> mst[ ])
{
    int i, j, min; float minw; long vk, vs;
    GraphEdgeMatrix<EdgeType> temp;
    for (i=0; i<numNodes-1; i++) //mst 中存放顶点 0 到其余各顶点的边
    {
        mst[i].SetStartNo(nodes[0].GetNo( ));
        mst[i].SetEndNo(nodes[i+1].GetNo( ));
        mst[i].SetWeight(edges[0][i+1].GetWeight( ));
    }
    for (i=0; i<numNodes-1; i++) //共选 n-1 条边
    {
        minw=INFINITY; min=-2;
        for (j=i; j<numNodes-1; j++)
            //从  $vk \in U, vs \in V-U$  的所有边中选择权值最小的边
            if (mst[j].GetWeight( )<minw) { minw=mst[j].GetWeight( ); min=j; }
            if (min==-2) { cout<<"Graph not connected!"<<endl; return false; }
            if (min!=i) { Swap(mst[min], mst[i]); }
        vk=mst[i].GetEndNo( );
        long vkPos, vsPos;
        for (j=i+1; j<numNodes-1; j++) //调整 mst[i+1] 到 mst[n-1] 中
        {
            vs=mst[j].GetEndNo( );
            LocateVertexNo(vk, vkPos); LocateVertexNo(vs, vsPos);
            minw=edges[vkPos][vsPos].GetWeight( );
            if (minw<mst[j].GetWeight( ))
                { mst[j].SetWeight(minw); mst[j].SetStartNo(vk); } //if
        } //for
    }
```

```

    } //for
    return true;
} //prim

```

对图 11.7 (a) 中的网, 利用程序 11-10 的算法依次输出生成树上的 5 条边分别为  $\{(v_1, v_3), (v_3, v_6), (v_6, v_4), (v_3, v_2), (v_2, v_5)\}$ 。分析上面算法, 普里姆算法的时间复杂度为  $O(n^2)$ , 与网中的边数无关, 因此该算法适于求边稠密网的最小生成树。

## 2. 克鲁斯卡尔 (Kruskal) 算法

该算法按照另一种方法来求网的最小生成树。假设连通网  $N=(V, \{E\})$ , 则令最小生成树的初始状态为只有  $n$  个顶点而无边的非连通图  $T=(V, \{\})$ , 图中每个顶点自成一个连通分量。在  $E$  中选择代价最小的边, 若该边依附的顶点落在  $T$  中不同的连通分量上, 则将此边加入到  $T$  中, 否则舍去此边而选择下一条代价最小的边, 其余类推, 直至  $T$  中所有顶点都在同一连通分量上为止。

用 Kruskal 算法得到最小生成树的过程如图 11.7 所示。其中代价最小的 4 条边分别为 2, 4, 5, 6, 它们被先后加入到  $T$  中, 代价为 7 的有两条边  $(v_1, v_4)$  和  $(v_2, v_3)$ , 其中, 边  $(v_1, v_4)$  的两个顶点在同一连通分量上, 它加入  $T$  中后会使  $T$  中产生回路, 所以舍去, 而边  $(v_2, v_3)$  满足条件, 因此将其加入  $T$  中, 由此构造出一棵最小生成树。

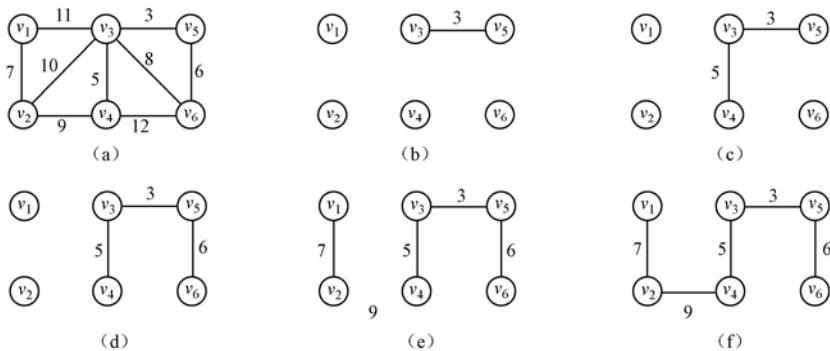


图 11.7 Kruskal 算法构造最小生成树的过程

Kruskal 算法的时间复杂度为  $O(elge)$  ( $e$  为网中边的数目), 因此它适于求边稀疏的网的最小生成树。采用 Kruskal 算法的最小生成树生成算法留给读者自行完成。

## 习题 11

11-1 关于找零钱问题, 假设售货员只有有限的 25 分, 10 分, 5 分和 1 分的硬币, 给出一种找零钱的贪婪算法。问: 这种方法总能找出具有最少硬币数的零钱吗? 证明结论。

11-2 已知  $n$  个任务的执行序列。假设任务  $i$  需要  $t_i$  个时间单位。若任务完成的顺序为  $1, 2, \dots, n$ , 则任务  $i$  完成的时间为  $c_i = \sum_{j=1}^i t_j$ 。任务的平均完成时间 (average completion time, ACT) 为  $\frac{1}{n} \sum_{j=1}^n c_j$ 。

(1) 考虑有 4 个任务的情况, 每个任务所需时间分别是 (4, 2, 8, 1)。若任务的顺序为 1, 2, 3, 4, 则 ACT 是多少?

(2) 若任务顺序为 2, 1, 4, 3, 则 ACT 是多少?

(3) 创建具有最小 ACT 的任务序列的贪婪算法分  $n$  步来构造该任务序列, 在每一步中, 从剩下的任务里选择时间最小的任务。对于 (1), 利用这种策略获得的任务顺序为 4, 2, 1, 3, 这种顺序的 ACT 是多少?

(4) 写一个 C++ 程序实现 (3) 中的贪婪策略, 程序的复杂性应为  $O(n \lg n)$ , 试证明之。

11-3 编写 C++ 程序实现 0-1 背包问题, 使用如下启发式方法: 按价值密度非递减的顺序打包。

11-4 如果第①步选择了顶点 1, 给出图 11.6 的工作过程。

11-5 对于二分图覆盖问题设计另外一种贪婪启发式方法, 可使用如下贪婪准则: 如果  $B$  中的某一个顶点仅被  $A$  中一个顶点覆盖, 则选择  $A$  中这个顶点; 否则, 从  $A$  中选择一个顶点, 使得它所覆盖的未被覆盖的顶点数目最多。

(1) 给出这种贪婪算法的伪代码。

(2) 编写一个 C++ 函数作为 `Undirected` 类的成员来实现上述贪婪算法。

(3) 函数的复杂度是多少?

11-6 令  $G$  为无向图,  $S$  为  $G$  中顶点的子集, 当且仅当  $S$  中的任意两个顶点都有一条边相连时,  $S$  为完备子图。完备子图的大小即为  $S$  中的顶点数目。最大完备子图即为具有最大顶点数目的完备子图。在图中寻找最大完备子图的问题 (即最大完备子图问题) 是一个 NP-完全问题。

(1) 给出最大完备子图问题的一种可行的贪婪算法及其伪代码。

(2) 给出一个能用 (1) 中的启发式算法求解最大完备子图的图例, 以及不能用该算法求解的一个图例。

(3) 将 (1) 中的启发式算法实现为 `BMCover::Clique(int C, int m)` 共享成员, 其中最大完备子图的大小返回到  $m$  中, 最大完备子图的顶点返回到  $C$  中。

(4) 代码的复杂度是多少?

11-7 证明当按路径长度的顺序产生一条最短路径时, 所产生的下一条最短路径总是由已产生的一条最短路径扩充一条边得到。

## 第 12 章 分而治之算法

分而治之是一种使用非常广泛的算法设计方法，其思想早在远古时代就被君主和殖民者们用于阶级统治，它是一种对复杂问题进行分解并逐一解决的方法，在算法设计中也常用到。本章将重点介绍分而治之方法在解决最小最大问题、排序问题、选择问题及计算几何问题等方面的应用。

### 12.1 算 法 思 想

分而治之的思想类似于软件的模块化设计方法，它将一个大问题分解成多个小问题，然后由小问题的解获得大问题的解。通常，由分而治之的算法所得到的问题与原来的大问题具有相同的类型，因此当分解后的小问题相对而言还是较大时，可根据需要对小问题继续采用分而治之的方法来求解。由此可见，分而治之算法可通过递归方式来实现。

分而治之算法的实现通常包括两个方面的内容：①基值处理部分，即问题分解到足够小后需要进行的处理；②问题分解部分，该部分又包括递归调用和合并处理两部分。这几个处理环节在很多情况下是相互渗透且合而为一的，但递归过程还是能够明显地与其他处理过程相区别的。

分而治之算法的重点在于分割原则的确定，不同的分割原则会导致方法的执行效率不同，这一点需要结合具体的应用来分析。根据大量的实践结果可得出，在采用该方法时，如果每次分割后的子问题规模大致相当，则算法的执行效果较好。下面将结合一些具体的例子来讨论分而治之算法在解决问题过程中的思路和方法。

### 12.2 应 用

#### 12.2.1 最大最小问题

**【例 12-1】** 一个老板有一袋重量不同的金币，每个月将两个金币分别奖励给表现优异的两名雇员，排名第一的雇员得到袋中最重的金币，而排名第二的雇员得到袋中最轻的金币。如果有新的金币周期性地加入袋中，则每个月都必须找出最轻和最重的金币。现在，需要找到一种方法来实现用最少的比较次数找出最轻和最重的金币。

比较的方法有很多种，假设袋中有  $n$  个金币，可通过  $n-1$  次比较找到最重的金币，然后再从余下的  $n-1$  个金币中用类似的方法通过  $n-2$  次比较找出最轻的金币，这种方法的比较总次数为  $2n-3$ 。

下面采用分而治之的方法对这个问题进行求解。当  $n$  很小时，如  $n \leq 2$ ，只需一次比较就能分析出最重和最轻的金币；当  $n$  较大时 ( $n > 2$ )，把这袋金币平均分成两个小袋  $A$  和  $B$ ，然后分别找出在  $A$  和  $B$  中最重和最轻的金币。设  $A$  和  $B$  中最重和最轻的金币分别为  $H_A$  与  $L_A$ 、 $H_B$  和  $L_B$ ，则通过分别比较  $H_A$  和  $H_B$ 、 $L_A$  和  $L_B$ ，可找到所有金币中最重和最轻的。

若  $A$  和  $B$  中的金币个数大于 2，则可递归地应用分而治之的算法。

假设袋子中的金币数为 8，则这个袋子被平分为各有 4 个金币的两个袋子  $A$  和  $B$ 。为了在  $A$  中找出最重和最轻的金币， $A$  中的 4 个金币被分成两组  $A_1$  和  $A_2$ 。每一组有 2 个金币，可用一次比较在  $A_1$  中找出较重的金币  $H_{A1}$  和较轻的金币  $L_{A1}$ 。经过另外一次比较，又能在  $A_2$  中找出  $H_{A2}$  和  $L_{A2}$ 。通过比较  $H_{A1}$  和  $H_{A2}$ ，能找出  $H_A$ ；通过比较  $L_{A1}$  和  $L_{A2}$  找出  $L_A$ 。这样，通过 4 次比较可找到  $H_A$  和  $L_A$ 。同样，在  $B$  中需要进行 4 次比较来确定  $H_B$  和  $L_B$ ，最后通过比较  $H_A$  和  $H_B$  ( $L_A$  和  $L_B$ ) 便得到了所有金币中最重和最轻的。因此，当  $n=8$  时，这种分而治之的方法需要进行 10 次比较。

设分而治之方法所需要的比较次数为  $c(n)$ ，为方便起见，假设  $n$  是 2 的幂。当  $n=2$  时， $c(n)=1$ ；当  $n>2$  时， $c(n)=2c(n/2)+2$ 。若  $n$  是 2 的幂，采用迭代计算可知  $c(n)=3n/2-2$ ，由此可见，分而治之的方法比逐一比较方法要更有效。在有些情况下，采用一个非递归程序来完成分而治之的算法得到结果的速度会比递归方式更快。上述金币问题的分而治之算法（例 12-1）就可通过一个非递归程序来更快地完成。

例 12-1 中寻找 8 个金币中最轻和最重金币的过程可用图 12.1 所示的二叉树来表示。这棵树的叶子分别表示 8 个金币 ( $a, b, \dots, h$ )，每个阴影结点表示一个包含其子树中所有叶子的问题。因此，根结点  $A$  表示寻找 8 个金币中最轻、最重金币的问题，而结点  $B$  表示找出  $a, b, c$  和  $d$  这 4 个金币中最轻和最重金币的问题。算法从根结点开始。由根结点表示的 8 金币问题被划分成由结点  $B$  和  $C$  所表示的两个 4 金币问题。在  $B$  结点上，4 金币问题被划分成由  $D$  和  $E$  所表示的 2 金币问题。可通过比较金币  $a$  和  $b$  中哪一个较重来解决  $D$  结点所表示的 2 金币问题。在解决了  $D$  和  $E$  所表示的问题之后，可通过比较  $D$  和  $E$  中所找到的轻金币和重金币来解决  $B$  表示的问题。接着，在  $F, G$  和  $C$  上重复这一过程，最后解决问题  $A$ 。

可将递归的分而治之算法分成以下步骤执行。

① 从图 12.1 中的二叉树由根至叶的过程中把一个大问题分成多个小问题，小问题的大小为 1 或 2。

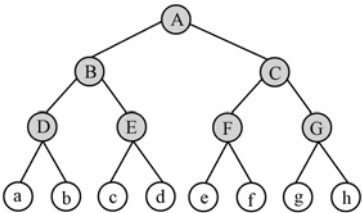


图 12.1 找出 8 个金币中最轻和最重的金币

② 比较每个大小为 2 的问题中的金币，确定哪一个较重和哪一个较轻。在结点  $D$ 、 $E$ 、 $F$  和  $G$  上完成这种比较。大小为 1 的问题中只有一个金币，它既是最轻的金币也是最重的金币。

③ 对较轻的金币进行比较以确定哪一个金币最轻，对较重的金币进行比较以确定哪一个金币最重。对于结点  $A \sim C$  执行这种比较。

根据上述步骤，可得出非递归代码，见程序 12-1。该程序用于寻找到数组  $w[0:n-1]$  中的最小数和最大数，若  $n<1$ ，则程序返回 false，否则返回 true。



```
template<class T>
bool MinMax(Tw[ ], int n, T& Min, T& Max)
{//寻找 w[0:n-1] 中的最小和最大值, 如果少于一个元素, 则返回 false, 特殊情形: n≤1
    if (n<1) return false; if (n==1) { Min=Max=0; return true; } //if
//对 Min 和 Max 进行初始化
    int s; //循环起点
    if (n%2) { Min=Max=0; s=1; } //if n 为奇数
    else { //n 为偶数, 比较第一对
        if (w[0]>w[1]) { Min=1; Max=0; } //if
        else { Min=0; Max=1; } //else
        s=2; } //else
//比较余下的数对
    for (int i=s; i<n; i+=2) {
//寻找 w[i] 和 w[i+1] 中的较大者, 然后将较大者与 w[Max] 进行比较, 将较小者与 w[Min] 进行比较
        if (w[i]>w[i+1]) {
            if (w[i]>w[Max]) Max=i;
            if (w[i+1]<w[Min]) Min=i+1; } //if
        else { if (w[i+1]>w[Max]) Max=i+1;
            if (w[i]<w[Min]) Min=i; } //else
    } //for
    return true;
} //MinMax
```

对上述程序进行复杂度分析, 当  $n$  为偶数时, 在 for 循环外部将执行一次比较, 而在 for 循环内部执行  $3(n/2-1)$  次比较, 比较的总次数为  $3n/2-2$ ; 当  $n$  为奇数时, for 循环外部没有执行比较, 而内部执行了  $3(n-1)/2$  次比较。因此无论  $n$  为奇数或偶数, 当  $n>0$  时, 比较的总次数为  $[3n/2]-2$ 。

## 12.2.2 归并排序

分而治之方法可用来解决排序问题。所谓排序, 是指将  $n$  个元素排成非递减顺序。分而治之方法通常采用以下的步骤来进行排序: 若  $n$  为 1, 则算法终止; 否则将这一元素集合分割成两个或更多个子集合, 对每一个子集合分别排序, 然后将排好序的子集合归并为一个集合。假设含有  $n$  个元素的集合仅被分为  $A$ 、 $B$  两个子集, 则会出现多种子集合的划分。

一种方法是把前面  $n-1$  个元素放到  $A$  中, 最后一个元素放到  $B$  中, 由于  $B$  中仅含一个元素, 因此只需对  $A$  按照同样的方式进行递归排序, 最后将排序后的  $A$  和  $B$  进行合并。这种方法实际上就是插入排序的递归算法, 该算法的复杂度为  $O(n^2)$ 。

另一种方法是将含有最大值的元素放入  $B$  中, 剩下的放入  $A$  中, 然后  $A$  被递归排序, 最后将  $B$  添加到  $A$  中进行合并。这种方法实际上为选择排序的递归算法, 其算法的复杂度也为  $O(n^2)$ 。

若采用冒泡过程来寻找最大值并把它移到最右边的位置, 则该方法就是冒泡排序的递

归算法。

由此可看出，上述的分割方案都是将  $n$  个元素分成两个极不平衡的集合  $A$  和  $B$ 。如果对  $n$  个元素每次都采用平衡分割方法进行排序，是否会提高算法效率呢？先看下面的例子。

**【例 12-2】** 对一个包含 8 个元素的集合[30, 15, 6, 24, 28, 7, 19, 13]进行平衡分割排序。如果选定  $k=2$ ，则分成两个子序列[30, 15, 6, 24]和[28, 7, 19, 13]并对其独立排序，这两个子序列的排序过程仍然可递归采用平衡分割排序再合并的方法进行，排序后的子序列分别为[6, 15, 24, 30]和[7, 13, 19, 28]，然后从两个子序列的头部开始归并，得到最终排序序列[6, 7, 13, 15, 19, 24, 28, 30]。

设  $t(n)$  为分而治之的排序算法在最坏情况下所需花费的时间，它包含每趟归并过程需要进行的排序时间（也就是元素比较次数）以及需要进行归并的趟数。从分而治之算法的归并过程可看出，不管怎么划分待排序的集合，每趟归并过程都要对划分到不同子集合中的  $n$  个元素进行比较，因此需要的时间都为  $O(n)$ 。而归并趟数取决于每次分割后子集合的大小，当进行平衡分割时，所划分的子集合大小最接近，这样使得总的分割趟数最小，为  $O(\lg n)$ ，在此方式下，分而治之算法具有最佳性能，即  $t(n)=O(n \lg n)$ 。它实质上就是第 9 章介绍的 2 路归并排序算法。在此给出 2 路归并排序的另一种实现方法，见程序 12-2。

程序 12-2 2 路归并排序的另一种实现方法

```
template<class T>
void MergeSort(T a[ ],int n)
{ //使用归并排序算法对 a[0..n-1]进行排序
    T *b=new T [n]; int s=1; //段的大小
    while (s<n)
    { MergePass(a,b,s,n); //从 a 归并到 b
      s+=s; MergePass(b,a,s,n); //从 b 归并到 a
      s+=s; }
}
```

为了完成排序代码，首先需要完成函数 MergePass。函数 MergePass（见程序 12-3）仅用来确定欲归并子序列的左端和右端，实际的归并工作由函数 Merge（见程序 12-4）完成。函数 Merge 要求针对类型 T 定义一个操作符 “<=”。如果需要排序的数据类型是用户自定义类型，则必须重载操作符 “<=”。

程序 12-3 MergePass 函数

```
template<class T>
void MergePass(Tx[ ],Ty[ ],int s,int n)
{ //归并大小为 s 的相邻段
    int i=0;
    while (i<=n-2*s) //归并两个大小为 s 的相邻段
    { Merge(x,y,i,i+s-1,i+2*s-1); i=i+2*s; }
    if (i+s<n) Merge(x,y,i,i+s-1,n-1); //剩下不足 2 个元素
    else for (int j=i; j<=n-1; j++) y[j]=x[j]; //把最后一段复制到 y
}
```

---

```

template<class T>
void Merge(Tc[ ], Td[ ], int l, int m, int r)
{ //把 c[l:m] 和 c[m:r] 归并到 d[l:r]
    int i=l; //第一段的游标
    j=m+1; //第二段的游标
    k=l; //结果的游标
    //只要在段中存在 i 和 j, 就不断进行归并
    while ((i<=m)&&(j<=r))
        if (c[i]<=c[j]) d[k++]=c[i++]; else d[k++]=c[j++];
    //考虑余下的部分
    if (i>m) for (int q=j; q<=r; q++) d[k++]=c[q];
    else for (int q=i; q<=m; q++) d[k++]=c[q];
}

```

---

### 12.2.3 快速排序

分而治之思想也体现在第 9 章介绍过的快速排序中, 只是它在快速排序中的应用过程与在归并排序中的不一样。在快速排序中, 每次都从待排序集合中选择一个值作为中间值(称为轴值), 然后以该值为基准, 将原来的序列分割成左、右两个部分, 其中左边部分的元素均小于轴值, 右边部分的元素均大于或等于轴值, 由此能确定轴值在最终排序序列中的位置, 然后再按照此方法分别对左边部分和右边部分进行分割, 直到被分割的子序列长度均为 1, 这样便获得了所有待排序元素在有序序列中的位置, 从而完成排序。

快速排序算法的代价同样也取决于分而治之过程中的分割结果, 也就是取决于每次设定的轴值是否能将原来的序列分割成长度相当的左、右两个部分。在最坏情况下, 左边部分总是为空, 快速排序所需的计算时间为  $O(n^2)$ 。在最好情况下, 左、右两部分的元素数目大致相同, 快速排序的复杂度为  $O(n \lg n)$ 。可证明快速排序的平均复杂度也是  $O(n \lg n)$ 。

中值快速排序是对第 9 章中程序 9-8 所描述的快速排序算法的一种改进, 它将每趟排序过程中轴值的选择由序列的第一个元素改为选择序列中的第一个元素、中间元素以及最后一个元素这三者中大小居中的那个元素。因此, 这种算法有更好的平均性能。

### 12.2.4 选择问题

选择问题就是对于给定的  $n$  个元素的数组  $a[0:n-1]$ , 要求从中找出第  $k$  小的元素。当  $a[0:n-1]$  被排序时, 该元素就是  $a[k-1]$ 。选择问题的一个应用是寻找中值元素。中值在很多统计问题中都非常有用, 如查找中间年龄、中间成绩、中间高度、中间数量等。除了找中间值外, 其他的  $k$  值也有用, 例如, 一个包含  $n$  个元素的序列存储的是员工工资信息, 要求将该工资情况分成三个等级。解决的方法是, 寻找初始序列排序后的第  $n/3$  和第  $2n/3$  位置上的元素, 以它们为界就可获得工资的三个等级情况。选择问题的解决方法是: 首先利用分而治之的方法对初始序列进行排序, 然后从排序后的序列中取出第  $k$  个元素。若采用上述的快速排序或归并排序, 其平均复杂度都为  $O(n \lg n)$ 。

程序 12-5 给出了寻找第  $k$  个元素的实现过程。

---

```

#include "exception.h"
template<class T>
void Swap(T &x, T &y)
{
    T k;
    k=x; x=y; y=k;
}

template<class T>
T Select(Ta[ ], int n, int k)
{//返回 a[0:n-1] 中第 k 小的元素
//假定 a[n] 是一个伪最大元素
    if (k<1 || k>n) throw OutOfBounds( ); return select(a, 0, n-1, k);
}

template<class T>
T select(T a[ ], int l, int r, int k)
{//在 a [l:r] 中选择第 k 小的元素
    if (l>=r) return a[l];
    int i=l; //从左至右的游标
    j=r+1; //从右到左的游标
    T pivot=a[l];
    //把左侧大于等于 pivot 的元素与右侧小于等于 pivot 的元素进行交换
    while (true) {
        do {//在左侧寻找大于等于 pivot 的元素
            i=i+1;
        } while (a[i]<pivot);
        do {//在右侧寻找小于等于 pivot 的元素
            j=j-1;
        } while (a[j]>pivot);
        if (i>=j) break; //未发现交换对象
        Swap(a[i], a[j]);
    }

    if (j-l+1==k) return pivot;
    a[l]=a[j]; a[j]=pivot; //设置 pivot
    //对一个段进行递归调用
    if (j-l+1<k) return select(a, j+1, r, k-j+1-1);
    else return select(a, l, j-1, k);
}

```

---

可以看出，程序 12-5 是在第 9 章的快速排序算法基础上进行了修改。在最坏情况下，该程序的复杂度是  $O(n^2)$ ，此时分割后的左边部分子序列总是为空，而且第  $k$  个元素总是位于右边部分的子序列中。如果左边部分和右边部分总是大小相同或相差不超过一个元素，假定  $n$  是 2 的幂，那么通过使用迭代方法，可得到时间代价  $t(n) = O(n)$ 。如果仔细选择轴值元

素，则最坏情况下的时间开销也可变成  $O(n)$ 。

一种选择轴值元素的方法是使用“中间的中间”规则，该规则首先将数组  $a$  中的  $n$  个元素分成  $n/r$  组 ( $r$  为某一整常数)，除了最后一组外，每组都有  $r$  个元素；然后，通过在每组中对  $r$  个元素进行排序来寻找每组中位于中间位置的元素；最后，根据所得到的  $n/r$  个中间元素，递归使用选择算法，求得所需要的轴值元素。

### 12.2.5 距离最近的点对问题

距离最近的点对问题就是对于给定的  $n$  个点  $(x_i, y_i)$  ( $1 \leq i \leq n$ )，要求找出其中距离最近的两个点。

**【例 12-3】** 假设在一片金属上钻有  $n$  个大小一样的洞，如果洞之间距离太近，金属可能会断。若知道任意两个洞的最小距离，可估计金属断裂的概率。这种最小距离问题实际上也就是距离最近的点对问题。

通过检查所有的  $n(n-1)/2$  对点，并计算每一对点的距离，可找出距离最近的一对点。这种方法被称为直接法，需要的时间为  $O(n^2)$ 。采用分而治之求解算法的伪代码如图 12.2 所示。该算法对于小的问题采用直接方法求解，而对于大的问题，则首先把它划分为两个较小的问题  $A$  和  $B$ ，其大小均为  $\lfloor n/2 \rfloor$ 。

对于上述问题，初始时最近的两个点对可能都在  $A$  或  $B$  中，即落在同一子点集中，也可能一点在  $A$  中而另一点在  $B$  中，即分别落在不同子点集中。前一种情况可对  $A$  或  $B$  进行递归求解，而后一种情况需要采用一种不同的方法，该方法取决于点集是如何被划分的。一个合理的划分方法是从  $x_i$  (中间值) 处划一条垂线，线左边的点属于  $A$ ，线右边的点属于  $B$ 。位于垂线上的点可在  $A$  和  $B$  之间分配，以便满足  $A$ 、 $B$  的大小。下面通过几个例子来讨论对后一种点对分布情况的处理。

```
if (n较小) {用直接法寻找最近点对;
            Return ;}

//n较大
将点集分成大致相等的两个部分A和B;
确定A和B中的最近点对;
确定一点在A中、另一点在B中的最近点对;
从上面得到的三点对中，找出距离最小的一对点;
```

图 12.2 寻找最近的点对算法伪代码

**【例 12-4】** 考察图 12.3 (a) 中从  $a$  到  $n$  的 14 个点。这些点标绘在图 12.3 (b) 中。选取中点  $x_i=2$ ，垂线  $x=2$ ，如图 12.3 (b) 中的虚线所示。虚线左边的点 (如  $a, c, e, i, k$ ) 属于  $A$ ，右边的点 (如  $b, f, g, h, l, n$ ) 属于  $B$ 。 $d, m, i$  落在垂线上，可将  $d, m$  加入  $A$ ， $i$  加入  $B$ ，以便  $A$ 、 $B$  中包含的点数相同。

设  $\delta$  是  $A$  的最近点对和  $B$  的最近点对中较小的距离。若上述的后一种最近点对分布情况下的最近点对距离比  $\delta$  小，则每一个点距垂线的距离必小于  $\delta$ ，这样就可淘汰那些距垂线距离大于等于  $\delta$  的点，这样以便确定是否存在第二类点对 (对应于后一种情况)，其距离小于  $\delta$ 。

用  $RA$ 、 $RB$  分别表示  $A$  和  $B$  中剩下的点。如果存在点对  $(p, q)$ ， $p \in A$ ， $q \in B$  且  $p, q$  的距离小于  $\delta$ ，则  $p \in RA$ ， $q \in RB$ 。可通过每次检查  $RA$  中的一个点来寻找这样的点对。假设考察  $RA$  中的  $p$  点， $p$  的  $y$  坐标为  $p.y$ ，那么只需检查  $RB$  中满足  $p.y - \delta < q.y < p.y + \delta$  的  $q$

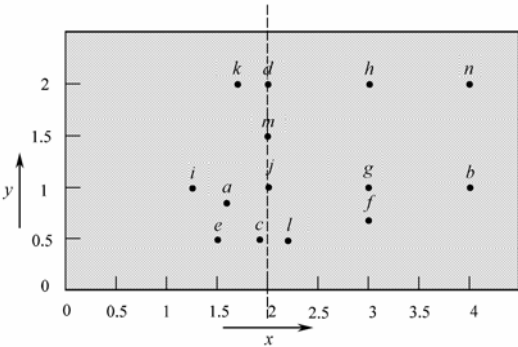
点，看是否存在与  $p$  间距小于  $\delta$  的点，这个  $\delta \times 2$  区域被称为  $p$  的比较区。

**【例 12-5】** 考察例 12-4 中的 14 个点， $A$  中的最近点对为  $(a, e)$ ，其距离约为 0.316。 $B$  中最近点对为  $(f, g)$ ，其距离为 0.3，因此  $\delta = 0.3$ 。当考察是否存在第二类点时，除  $d, j, c, l, m$  以外的点均被淘汰，因为它们距分割线  $x = 2$  的距离大于等于  $\delta$ 。 $RA = \{d, c, m\}$ ， $RB = \{i, l\}$ ，由于  $d$  和  $m$  的比较区中没有点，只需考察  $c$  即可。 $c$  的比较区中仅含点  $l$ 。计算  $c$  和  $l$  的距离，发现它小于  $\delta$ ，因此  $(c, l)$  是最近的点对。

分而治之算法的实现需要先确定什么是“小问题”以及如何来表示点。由于集合中少于两点时不存在最近点对，因此分解过程中应保证不会产生少于两点的点集，因此将少于 4 点的点集作为“小问题”，就可避免产生少于两点的点集。点可用 `Point1` 类来表示（见程序 12-6），其中每个点有 3 个参数：标号， $x$  坐标和  $y$  坐标。其中，标号为整数。

点	$a$	$b$	$c$	$d$	$e$	$f$	$g$	$h$	$i$	$j$	$k$	$l$	$m$	$n$
$x_i$	1.6	4	1.9	2	1.5	3	3	3	1.25	2	1.7	2.2	2	4
$y_i$	0.8	1	0.5	2	0.5	0.7	1	2	1	1	2	0.5	1.5	2

(a) 14 个点的位置信息



(b) 点的分布情况

图 12.3 14 个点的位置信息及其分布

程序 12-6 点类

```
class Point2;
class Point1 {
    friend float dist(const Point1&, const Point1&);
    friend void close(Point1 *, Point2 *, Point2 *, int, int, Point1&, Point1&, float&);
    friend bool closest(Point1 *, int, Point1&, Point1&, float&);
    friend void main();
public:
    int operator<=(Point1 a) const
    { return(x<=a.x); }
private:
    int ID; //点的编号
    float x, y; //点坐标
};
class Point2{
    friend float dist(const Point2&, const Point2&);
```

```

        friend void close(Point1 *, Point2 *, Point2 *, int, int, Point1&, Point1&, float&);
        friend bool closest(Point1 *, int, Point1&, Point1&, float&);
        friend void main( );

public:
    int operator<=(Point2 a) const
    { return (y<=a.y); }

private:
    int p; //数组 x 中相同点的索引
    float x, y; //点坐标
} ;

```

---

按  $y$  坐标排序的点保存在另一个使用类 **Point2**（见程序 12-7）的数组中，为了便于按  $y$  坐标排序，该类也已重载了操作符“ $\leq$ ”。成员  $p$  用于指向  $x$  中的对应点。程序 12-7 定义了一个模板函数 **dist** 来计算点  $a, b$  之间的距离。

#### 程序 12-7 计算两点之间的距离

---

```

template<class T>
inline float dist(const T& u, const T& v)
{ //计算点 u 和 v 之间的距离
    float dx=u.x-v.x ; float dy=u.y-v.y ;
    return sqrt(dx * dx+dy * dy);
}

```

---

如果点的数目少于两个，则函数 **closest**（见程序 12-8）返回 **false**，否则返回 **true**。当函数成功时，在参数  $a$  和  $b$  中返回距离最近的两个点，在参数  $d$  中返回距离。代码首先验证至少存在两个点，然后使用 **MergeSort** 排序算法（见程序 12-2），最后调用函数 **close**（见程序 12-9）来完成最近点对的求解。

#### 程序 12-8 预处理及调用 close

---

```

bool closest(Point1 X[ ], int n, Point1& a, Point1& b, float& d)
{ //在  $n \geq 2$  个点中寻找最近点对
  //如果少于 2 个点，则返回 false
  //否则，在 a 和 b 中返回距离最近的两个点
  if (n<2) return false;
  MergeSort(X, n); //按 x 坐标排序
  Point2 *Y=new Point2 [n]; //创建一个按 y 坐标排序的点数组
  for (int i=0; i<n; i++) { //将点 i 从 X 复制到 Y
    Y[i].p=i; Y[i].x=X[i].x; Y[i].y=X[i].y; }
  MergeSort(Y, n); //按 y 坐标排序
  Point2 *Z=new Point2 [n]; //创建临时数组
  close(X, Y, Z, 0, n-1, a, b, d); //寻找最近点对
  delete [ ] Y; delete [ ] Z; return true; //删除数组并返回
}

```

---

```

void close(Point1 X[ ], Point2 Y[ ], Point2 Z[ ], int l, int r, Point1& a, Point1&b,
float& d)
{ //X[1:r] 按 x 坐标排序
  //Y[1:r] 按 y 坐标排序
  if (r-l==1) { a=X[l]; b=X[r]; d=dist(X[l], X[r]); return; } //两个点
  if (r-l==2) { //三个点
    //计算所有点对之间的距离
    float d1=dist(X[l], X[l+1]); float d2=dist(X[l+1], X[r]);
    float d3=dist(X[l], X[r]);
    //寻找最近点对
    if (d1<=d2 && d1<=d3) { a=X[l]; b=X[l+1]; d=d1; return ; }
    if (d2<=d3) { a=X[l+1]; b=X[r]; d=d2; }
    else { a=X[l]; b=X[r]; d=d3; }
    return; }
  //多于三个点, 划分为两部分
  int m=(l+r)/2; //X[1:m] 在 A 中, 余下的在 B 中
  //在 Z[1:m] 和 Z[m+1:r]中创建按 y 排序的表
  int f=1; g=m+1; //Z[1:m]的游标, Z[m+1:r]的游标
  for (int i=1; i<=r; i++) if (Y[i].p>m) Z[g++]=Y[i]; else Z[f++]=Y[i];
  //对以上两个部分进行求解
  close(X, Z, Y, l, m, a, b, d);
  float dr; Point1 ar, br;
  close(X, Z, Y, m+1, r, ar, br, dr);
  //(a, b)是两者中较近的点对
  if (dr<d) { a=ar; b=br; d=dr; }
  Merge(Z, Y, l, m, r); //重构 Y
  //距离小于 d 的点放入 Z 中
  int k=1; //Z 的游标
  for (i=1; i<=r; i++) if (fabs(Y[m].x-Y[i].x)<d) Z[k++]=Y[i];
  //通过检查 Z[1:k-1]中的所有点对, 寻找较近的点对
  for (i=1; i<k; i++) {
    for (int j=i+1; j<k && Z[j].y-Z[i].y<d; j++) { float dp=dist(Z[i], Z[j]);
      if (dp<d) { d=dp; a=X[Z[i].p]; b=X[Z[j].p]; } //较近的点对
    }
  }
}
}
}

```

下面分析该问题处理算法的复杂度。令  $t(n)$  代表处理  $n$  个点时函数 `close` 所需要的时间。当  $n < 4$  时,  $t(n)$  等于某个常数  $d$ 。当  $n \geq 4$  时, 需花费  $O(n)$  时间来完成以下工作: 将点集划分为两个部分, 两次递归调用后重构  $Y$ , 淘汰距分割线很远的点, 寻找更好的第二类点对。两次递归调用需分别耗时  $t(\lceil n/2 \rceil)$  和  $t(n/2)$ , 因而可通过递归方式求得  $t(n) = O(n \lg n)$ 。此外, 函数 `closest` 还需要耗时  $O(n \lg n)$  来完成如下额外工作: 对  $X$  进行排序, 创建  $Y$  和  $Z$ , 对  $Y$  进行排序。因此, 分而治之最近点对求解算法的时间复杂度为  $O(n \lg n)$ 。



# 习题 12

12-1 顺序统计是指在  $n$  个数的集合中（如一维数组），找出第  $k$  小的数（即从小到大排行第  $k$  的，最小元素为第 1 小元素）。一个直观的解法是，将这几个数排序，然后取其中第  $k$  个即可。但是，在一般情况下，排序的时间复杂度不能低于  $O(n \lg n)$ 。试不通过排序直接求解，而是通过分而治之算法求解。

12-2 设有  $n$  个篮球队参加循环赛，共进行  $n-1$  天比赛，每队每天必须参加且仅参加一场比赛，这里设  $n=2^k$ （ $k$  为正整数）。试证明：当  $n$  满足前述条件时，该问题有解。

12-3 用穷举法生成  $n$  阶魔方。所谓  $n$  阶魔方，是指一个  $n \times n$  方阵，将  $n^2$  个自然数  $1 \sim n^2$  分别填入  $n^2$  个不同的格子中，它的每行之和、每列之和、每个对角线或反对角线之和都分别等于某个常数。3 阶魔方的例子如图 12.4 所示。

4	9	2
3	5	7
8	1	6

图 12.4 3 阶魔方的例子

12-4 编写生成组合的程序。例如，从  $(a,b,c,d)$  中取 3 个字母的不同取法或组合有  $abc, bcd, acd$  等。

12-5 编写程序，在由  $n$  个数构成的序列中，找出最长的单调递增子序列。要求：时间复杂度不超过  $O(n^2)$ 。

# 第13章 动态规划

动态规划是本书介绍的 5 种算法设计方法中难度最大的一种，它建立在最优原则的基础上。采用动态规划方法，可高效地解决许多用贪婪算法或分而治之算法无法解决的问题。在介绍动态规划的基本思想之后，本章将分别讨论动态规划方法在解决背包问题、图像压缩、矩阵连乘等方面的应用。

## 13.1 算 法 思 想

动态规划和贪婪算法类似，也是将问题的求解过程看成一系列决策的过程。不同的是，贪婪算法中每一次根据贪婪准则做出的决策是一个不可撤回的决策，而动态规划中的每一步决策还要考察每个最优决策序列中是否包含一个最优子序列，因此，动态规划通常用于求解一个问题在某种意义下的最优解。下面通过一个例子来进行说明。

**【例 13-1】** 在如图 13.1 所示的有向图中要寻找一条从源点  $s=1$  到目标点  $d=5$  的最短路径。从源点 1 出发，接下来可选择结点 2, 3 或 4。若选择了结点 3，则下面要求解的问题变成选择一条从 3 到 5 的最短路径，如果 3 到 5 的路径不是最短的，则从 1 开始经过 3 和 5 的路径也不会是最短的。例如，若选择的子路径为 3, 2, 5，则 1 到 5 的路径为 1, 3, 2, 5，其耗费为 11；而若选择最短子路径为 3, 4, 5，则 1 到 5 的路径为 1, 3, 4, 5，其耗费为 9。显然，前者的耗费更大。由此可见，在最短路径问题中，如果在某一次决策时到达了某个结点  $v$ ，那么在后面选择从  $v$  到  $d$  的路径时，都必须采用最优策略。

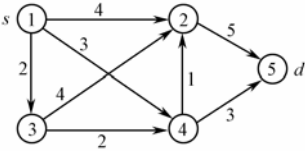


图 13.1 一个有向图

动态规划方法采用最优原则来建立用于计算最优解的递归式。所谓最优原则，是指在问题的求解过程中，当通过某种策略执行到某一步时，不管这一步之前的决策是否是最优，这一步之后的决策必须是基于当前状态下的最优决策。由于有些问题的某些递归式不一定能保证最优原则，因此在求解过程中需要对它进行验证。若不能保持最优原则，则不可应用动态规划方法。在得到最优解的递归式之后，需要执行回溯以构造最优解。虽然在求解动态规划问题时用递归程序来实现比较方便，但由于求解过程中的重复计算非常多，如果不能有效地避免重复计算，则递归程序的复杂度将非常大。同时，动态规划递归方程也可用迭代方式来求解，尽管在复杂度方面，迭代程序与避免重复计算的递归程序大致相同，但迭代程序不需要附加的递归栈空间，因此将比避免重复计算的递归程序更快。

## 13.2 应 用

### 13.2.1 0-1 背包问题

#### (1) 递归策略

前面章节介绍的 0-1 背包问题需要确定的  $x_1$  到  $x_n$  的值，假设按照  $i$  从 1 到  $n$  的次序来

确定  $x_i$  的值。在第一次决策之后，剩下的问题便是考虑背包容量为  $r$ ,  $r \in \{c, c-w_1\}$  时的决策。不管  $x_1$  为 0 还是 1,  $[x_2, x_3, \dots, x_n]$  必须是第一次决策之后的一个最优方案，如果不是，则  $[x_1, x_2, \dots, x_n]$  也不会是总体的最优解。因此，最优决策序列由最优决策子序列组成。下面给出该问题的动态规划递归方程。

假设  $f(i, y)$  表示剩余容量为  $y$ ，剩余物品为  $i, i+1, \dots, n$  时的最优解的值，即：

$$f(n, y) = \begin{cases} p_n & y \geq w_n \\ 0 & 0 \leq y < w_n \end{cases} \tag{13-1}$$

和

$$f(i, y) = \begin{cases} \max\{f(i+1, y), f(i+1, y-w_i+p_i)\} & y \geq w_i \\ f(i+1, y) & 0 \leq y < w_i \end{cases} \tag{13-2}$$

$f(1, c)$  是初始时背包问题的最优解，可使用式（13-2）通过递归或迭代来求解  $f(1, c)$ 。从  $f(n, *)$  开始迭代， $f(n, *)$  先由式（13-1）得出，然后由式（13-2）递归计算  $f(i, *)$  ( $i = n-1, n-2, \dots, 2$ )，最后由式（13-2）得出  $f(1, c)$ 。另外， $x_i$  值的计算步骤如下：若  $f(1, c) = f(2, c)$ ，则  $x_1=0$ ，否则  $x_1=1$ 。接下来，需从剩余容量  $c-w_1$  中寻求最优解，用  $f(2, c-w_1)$  表示最优解，其余类推，可得到所有的  $x_i$  ( $i=1, 2, \dots, n$ ) 值。

程序 13-1 给出了根据以上的动态规划递归方程实现 0-1 背包问题的递归函数。

程序 13-1 0-1 背包问题的递归函数

```
int F(int i, int y)
{
    //返回 f(i, y)，此处 w[0..n-1] 为背包重量，p[0..n-1] 为价值
    if (i==n) return (y<w[n-1]) ? 0 : p[n-1];
    if (y<w[i-1]) return F(i+1, y);
    return max(F(i+1, y), F(i+1, y-w[i-1])+p[i-1]);
}
```

程序 13-1 的时间复杂度  $t(n)$  满足条件： $t(1) = a$ ； $t(n) \leq 2t(n-1)+b(n>1)$ ，其中  $a$ 、 $b$  为常数。通过求解可得  $t(n) = O(2^n)$ 。

(2) 权为整数的迭代方法

**【例 13-2】** 设  $n = 5$ ,  $w = [2, 2, 6, 5, 4]$  且  $c = 10$ ,  $p = [9, 5, 7, 6, 9]$ ，求  $f(1, 10)$ 。

$f(1, 10)$  通过调用函数  $F(1, 10)$  求得，递归调用的关系如图 13.2 的树型结构所示，其中每个结点的值用  $y$  值来标记。根结点为  $f(1, 10)$ ，它有左、右两个孩子，分别对应为  $f(2, 10)$  和  $f(2, 8)$ ，对于第  $j$  层的结点有  $i = j$ ，总共执行了 28 次递归调用。注意，其中可能含有重复前面工作的结点，如  $f(3, 8)$  计算过两次，相同情况的还有  $f(4, 8)$ 、 $f(4, 2)$ 、 $f(4, 6)$ 、 $f(5, 8)$ 、 $f(5, 3)$ 、 $f(5, 2)$ 、 $f(5, 6)$  和  $f(5, 1)$ （即图中的阴影结点），若这些前面的计算结果被保留，则可去掉这些阴影结点，因此结点数将减少至 19 个。

因此，可定义一个用于保留已被计算出的  $f(i, y)$  值的表格  $L$ ，该表格中的元素是三元组  $(i, y, f(i, y))$ 。在计算每一个  $f(i, y)$  之前，先检查表  $L$  中是否已包含一个三元组  $(i, y, *)$ ，其中， $*$  表示任意值。如果已包含，则从该表中取出  $f(i, y)$  的值；否则，对  $f(i, y)$  进行计算并将计算所得的三元组  $(i, y, f(i, y))$  加入表  $L$  中。根据以上分析的结果，可设计一个简单的算法来求解  $f(1, c)$ （见程序 13-2），设权为整数，此算法中对每个  $f(i, y)$  只计算一次，同时用二维数组  $f[i][j]$  来保存各  $f$  的值，回溯函数 Traceback 用于确定由程序 13-2 所产生的  $x_i$  值。



即不同像素用不同位数来存储。像素值为 0 和 1 时，需要 1 位存储空间；值为 2 和 3 时，需要 2 位；值 4, 5, 6 和 7 时，需要 3 位；其余类推。变长模式的实现步骤如下。

① 根据图 13.3 (a) 所示的折线方向将图像进行以行为主次序的蛇形线性化，将  $m \times m$  维图像转换为  $1 \times m^2$  维矩阵。

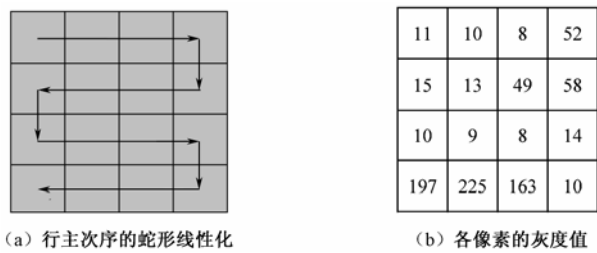


图 13.3 数字化图像

② 根据像素位数对其进行分段，具有相同值的像素分到同一段中，每个段是相邻像素的集合且每段最多含 256 个像素。若相同位数的像素超过 256，则用两个以上的段表示。

③ 建立三个文件，第一个文件（设文件名为 SegmentLength）包含上一步中所建段的长度（减 1），文件中各项均为 8 位长；第二个文件（设文件名为 BitsPerPixel）给出了各段中每个像素的存储位数（减 1），文件中各项均为 3 位；第三个文件（设文件名为 Pixels）则是以变长格式存储的像素的二进制串。

④ 压缩上一步所建立的文件，以减少空间需求。

**【例 13-3】** 对如图 13.3 (b) 所示的  $4 \times 4$  图像进行压缩。

① 按照蛇形线性化的行主次序，各像素的灰度值依次为 11, 10, 8, 52, 58, 49, 13, 15, 10, 9, 8, 14, 10, 163, 225 和 197，所需的位数分别为 4, 4, 4, 6, 6, 6, 4, 4, 4, 4, 4, 4, 4, 8, 8 和 8。

② 按等长的条件将像素进行分段，得到 [11, 10, 8], [52, 58, 49], [13, 15, 10, 9, 8, 14, 10] 和 [163, 225, 197] 4 个段。

③ 建立三个文件，其中 SegmentLength 的内容为 2, 2, 6, 2, BitsPerSegment 的内容为 3, 5, 3, 7; Pixels 包含了图像线性化后的 16 个灰度值，其中第 1~3 个各用 4 位进行存储，第 4~6 个各用 6 位进行存储，第 7~13 个各用 4 位进行存储，第 14~16 个各用 8 位进行存储，因此其包含的以变长格式存储的像素二进制串为：

1011 1010 1000 110100 111010 110001 1101 1111 1010 1001 1000 1110 1010 10100011  
11100001 11000101

这三个文件分别需要的存储空间为：文件 SegmentLength 需要 32 位，BitsPerSegment 需要 12 位，Pixels 需要 82 位，共 126 位。如果每个像素都用 8 位进行存储，则存储空间需要  $8 \times 16 = 128$  位，因此本例的图像采用变长格式存储节省了 2 位的空间。

下面考虑通过将某些相邻段进行合并的方式来进一步减少需要的存储空间。假设在完成上述的第②步之后产生了  $n$  个段，用段标题来存储段的长度以及该段中每个像素所占用的位数，每个段标题需 11 位。

现假设  $l_i$  和  $b_i$  分别表示第  $i$  段的段长和该段每个像素的长度，则存储第  $i$  段像素所需要的空间为  $l_i b_i$ ，总存储空间为  $\sum_{i=1}^n l_i b_i$ 。如果将段  $i$  和  $i+1$  进行合并，则合并后的段长应为  $l_i + l_{i+1}$ ，每个像素的存储位数为  $\max\{b_i, b_{i+1}\}$ 。尽管这种方式增加了文件 Pixels 的空间消耗，

但同时也减少了一个段标题的空间。

**【例 13-4】** 如果将例 13-3 中的第 1 段和第 2 段合并，合并后文件 SegmentLength 变为 5, 6, 2, BitsPerSegment 变为 5, 3, 7。而文件 Pixels 的前 36 位存储的是合并后的第 1 段：10111010100 0110100111010110001，其余的像素没有改变。因为减少了 1 个段标题，文件 SegmentLength 和 BitsPerPixel 的空间消耗共减少了 11 位，而文件 Pixels 的空间消耗增加 6 位，因此总共节约的空间为 5 位，空间总消耗为 121 位。

希望能设计一种算法使得在产生  $n$  个段之后能对相邻段进行合并，以便产生一个具有最小空间需求的新的段集合。在合并相邻段之后，可利用诸如 LZW 法或霍夫曼编码等其他技术来进一步压缩这三个文件。

令  $s_q$  为前  $q$  个段的最优合并所需要的空间。定义  $s_0 = 0$ 。考虑第  $i$  段 ( $i > 0$ )，假如在最优合并  $C$  中，第  $i$  段与第  $i-1, i-2, \dots, i-r+1$  段相合并，而不包括第  $i-r$  段。合并  $C$  所需要的空间消耗等于：

$$\text{第 1 段到第 } i-r \text{ 段所需空间} + l_{\text{sum}}(i-r+1, i) \times b_{\text{max}}(i-r+1, i) + l_1$$

式中， $l_{\text{sum}}(a, b) = \sum_{i=a}^b l_j$ ， $b_{\text{max}}(a, b) = \max\{b_a, \dots, b_b\}$ ， $l_1$  为第 1 段的长度值。

假如在  $C$  中第 1 段到第  $i-r$  段的合并不是最优合并，那么需要对合并进行修改，以使其具有更小的空间需求，即保证最优原则得以维持。故  $C$  的空间消耗为

$$s_i = s_{i-r} + l_{\text{sum}}(i-r+1, i) \times b_{\text{max}}(i-r+1, i) + l_1$$

式中  $r \in [1, i]$ ， $l_{\text{sum}} \leq 256$ （段长应小于  $2^8=256$ ）。尽管不知道如何选择  $r$ ，但知道要使  $C$  满足最小空间需求，因此在所有选择中， $r$  必须产生最小的空间需求。由此可产生递归式：

$$s_i = \min_{\substack{1 \leq k \leq i \\ l_{\text{sum}}(i-k+1, i) \leq 256}} \{s_{i-k} + l_{\text{sum}}(i-k+1, i) \times b_{\text{max}}(i-k+1, i)\} + l_1 \quad (13-3)$$

假定  $\text{kay}_i$  为取得最小值时  $k$  的值， $s_n$  为  $n$  段的最优合并所需要的空间，因此一个最优合并可通过  $\text{kay}$  的值构造出来。

**【例 13-5】** 假定在第②步中得到 5 个段，它们的长度为 [6, 3, 10, 2, 3]，像素位数为 [1, 2, 3, 2, 1]，若用式 (13-3) 计算  $s_n$ ，必须先求出  $s_{n-1}, \dots, s_0$  的值。已知  $s_0=0$ ，现计算  $s_1$  如下：

$$s_1 = s_0 + l_1 \times b_1 + l_1 = 17$$

$$\text{kay}_1 = 1$$

$s_2$  由下式得出：

$$s_2 = \min\{s_1 + l_2 b_2, s_0 + (l_1 + l_2) \times \max\{b_1, b_2\}\} + l_1 = \min\{17 + 6, 0 + 9 \times 2\} + 11 = 29$$

$$\text{kay}_2 = 2$$

其余类推，可得  $[s_1, s_2, \dots, s_5] = [17, 29, 67, 73, 82]$ ， $[\text{kay}_1, \text{kay}_2, \dots, \text{kay}_5] = [1, 2, 2, 3, 4]$ ，由  $s_5 = 82$  可知最优空间合并后需要 82 位的存储空间，同时由  $\text{kay}_5$  导出本合并的过程如下：因为  $\text{kay}_5 = 4$ ，所以  $s_5$  是由公式 (13-3) 在  $k = 4$  时取得的，因而最优合并包括段 1 到段  $(5-4) = 1$  的最优合并以及段 2, 3, 4 和 5 的合并，最后剩下段 1 及段 2 到段 5 的合并段。

(1) 递归方法

用递归式 (13-3) 可计算出  $s_i$  和  $\text{kay}_i$ ，程序 13-3 给出了该递归式计算的程序代码。

```
int *l; //l[1..n-1]存放长度, 0 号单元未用
int *b; //b[1..n-1]存放像素位, 0 号单元未用
int *kay; //kay[1..n-1]存放取得最小值时 k 的值
#define L 256
#define header 11
int S(int i)
{ //返回 S(i) 并计算 kay[i]
    if (i<0 ) return 0;
    //k=0 时, 根据式 (13-3) 计算最小值
    int lsum=l[i], bmax=b[i]; int s=S(i-1)+lsum * bmax; kay[i]=1;
    //对其余的 k 计算最小值并求取最小值
    for (int k=1; k<=i && lsum+l[i-k]<=L; k++) {
        lsum+=l[i-k];
        if (bmax<b[i-k]) bmax=b[i-k]; int t=S(i-k-1);
        if (s>t+lsum * bmax) { s=t+lsum * bmax; kay[i]=k+1; }
    }
    return s+header;
}
void Traceback(int kay[ ], int n)
{ //合并段
    if (n<0) return; Traceback(kay, n-kay[n]);
    cout<<"New segment begins at" <<(n-kay[n]+1)<<endl;
}
```

在上例中,  $n=6$  (0 号单元未用), 调用是使用  $S(n-1)$ ,  $Traceback(hay, n-1)$ 。分析以上程序的复杂度,  $t(0)=c$  ( $c$  为一个常数);  $t(n) \leq \sum_{j=\max(0, n-256)}^{n-1} t(j) + n$  ( $n>0$ ), 由递归方法可得  $t(n)=O(2^n)$ , 其中,  $Traceback$  的复杂度为  $O(n)$ 。

## (2) 无重复计算的递归方法

通过避免重复计算  $s_i$ , 可将函数  $S$  的复杂度减小到  $O(n)$ 。以例 13-5 中的 5 个段为例再进行进一步分析, 计算  $s_5$  时, 递归调用计算  $s_4, \dots, s_0$ ; 计算  $s_4$  时, 递归调用计算  $s_3, \dots, s_0$ , 因此  $s_4$  计算了一次, 而  $s_3$  计算了两次, 每一次计算  $s_3$  时都要计算一次  $s_2$ , 因此  $s_2$  共计算了 4 次, 而  $s_1$  则重复计算了 16 次。重复计算问题的解决可利用一个数组  $s$  来保存先前计算过的  $s_i$ , 改进后的代码见程序 13-4, 其中  $s$  为初值为 0 的全局整型数组。

## 程序 13-4 避免重复计算的递归算法

```
int *s;
int S1(int i)
{ //计算 S(i) 和 kay[i]
    //避免重复计算
    if (i==0) return 0; if (s[i]>0) return s[i]; //已计算完
```

```

//计算 s[i], 首先根据式 (13-3) 计算 k=1 时的最小值
    int lsum=l[i], bmax=b[i]; s[i]=S1(i-1)+lsum * bmax; kay[i]=1;
//对其他的 k 计算最小值并更新
    for (int k=2; k<=i && lsum+l[i-k+1]<=L; k++) {
        lsum+=l[i-k+1];
        if (bmax<b[i-k+1]) bmax=b[i-k+1]; int t=S(i-k);
        if (s[i]>t+lsum * bmax) { s[i]=t+lsum * bmax; kay[i]=k; }
    }
    s[i]+=header;
    return s[i];
}

```

---

上例中,  $n=6$  (0 号单元未用), 调用方式为  $S(n-1)$ 。使用分期计算模式来分析上述程序的时间复杂度。在该模式中, 总时间被分解为若干个不同项, 通过计算各项的时间然后求和来获得总时间。当计算  $s_i$  时, 若  $s_j$  还未算出, 则把调用  $S(j)$  的消耗计入  $s_j$ ; 若  $s_j$  已算出, 则把  $S(j)$  的消耗计入  $s_i$  (这里  $s_j$  依次把计算新  $s_q$  的消耗转移至每个  $s_q$ )。程序 13-4 的其他消耗也被计入  $s_i$ 。由于  $L$  是 256 之内的常数且每个  $l_i$  至少为 1, 所以程序 13-4 的其他消耗为  $O(1)$ , 即计入每个  $s_i$  的量是一个常数, 且  $s_i$  数目为  $n$ , 因此总的时代价为  $O(n)$ 。

### (3) 迭代方法

如果用式 (13-3) 依序计算  $s_1, s_2, \dots, s_n$ , 便可得到一个复杂度为  $O(n)$  的迭代方法 (见程序 13-5), 在该方法中, 在  $s_i$  计算之前,  $s_j$  必须已经计算好, 其中仍利用函数 Traceback (见程序 13-3) 来获得最优合并。

程序 13-5 迭代计算 s 和 kay

```

void Vbits (int l[ ], int b[ ], int n, int s[ ], int kay[ ])
{
//计算 s[i] 和 kay[i]
    int L=256, header=11; s[0]=0;
//根据式 (13-3) 计算 s[i]
    for (int i=1; i<=n; i++) {
        //k=1 时, 计算最小值
        int lsum=l[i], bmax=b[i]; s[i]=s[i-1]+lsum * bmax; kay[i]=1;
        //对其他的 k 计算最小值并更新
        for (int k=2; k<=i && lsum+l[i-k+1]<=L; k++) { lsum+=l[i-k+1];
            if (bmax<b[i-k+1]) bmax=b[i-k+1];
            if (s[i]>(s[i-k]+lsum*bmax)) {
                s[i]=s[i-k]+lsum*bmax; kay[i]=k; }
        }
        s[i]+=header;
    }
}

```

---



### 13.2.3 矩阵连乘法

矩阵连乘问题是指给定  $n$  个矩阵  $\{A_1, A_2, \dots, A_n\}$ , 其中  $A_i$  与  $A_{i+1}$  可相乘,  $i=1, 2, \dots, n-1$ , 由于矩阵乘法满足结合律, 因此这  $n$  个矩阵的连乘积  $A_1 A_2 \dots A_n$  可有多种不同的计算次序。虽然不同的计算次序得到的结果相同, 但在时间消耗上有很大的差距。

**【例 13-6】** 计算 3 个矩阵  $A$ 、 $B$  和  $C$  的乘积。计算过程可先用  $A$  乘以  $B$  得到矩阵  $D$ , 然后用  $D$  乘以  $C$  得到最终结果, 该执行过程记为  $(A \times B) \times C$ 。也可采用  $A \times (B \times C)$  的乘法次序。假定  $A$  为  $100 \times 1$  矩阵,  $B$  为  $1 \times 100$  矩阵,  $C$  为  $100 \times 1$  矩阵, 则  $A \times B$  的时间耗费为 10000, 得到的结果  $D$  为  $100 \times 100$  矩阵, 再与  $C$  相乘所需的时间耗费为 1000000, 因此计算  $(A \times B) \times C$  的总时间为 1010000, 同时还需 10000 个单元来存储  $A \times B$ 。若按照  $A \times (B \times C)$  的次序执行, 则  $B \times C$  的时间耗费为 10000, 得到的中间矩阵为  $1 \times 1$  矩阵, 再与  $A$  相乘的时间消耗为 100, 因此总的计算时间耗费为 10100, 并只需要 1 个单元来存储  $B \times C$ 。由此可知, 矩阵连乘问题的最优解是使计算耗费最小的执行次序。

上述例子中只是对三个矩阵进行连乘, 对于更多矩阵的连乘来说, 情况要复杂得多。例如, 计算矩阵乘积  $M_1 \times M_2 \times \dots \times M_k$ , 其中  $M_i$  是一个  $r_i \times r_{i+1}$  阶矩阵 ( $1 \leq i \leq k$ )。设  $k=4$ , 则矩阵运算  $A \times B \times C \times D$  可按以下几种次序进行计算  $A \times ((B \times C) \times D)$ ,  $A \times (B \times (C \times D))$ ,  $(A \times B) \times (C \times D)$ ,  $(A \times (B \times C)) \times D$ 。不难看出, 计算的方法数会随  $k$  以指数级方式增加, 因此对于  $k$  值很大的矩阵连乘问题而言, 考虑每一种计算次序并选择最优方法是不切实际的。

下面要介绍一种采用动态规划方法获得矩阵乘法次序的最优方法。这种方法可将算法的时间消耗降为  $O(q^3)$ 。用  $M_{ij}$  表示链  $M_i \times \dots \times M_j (i \leq j)$  的乘积。设  $c(i, j)$  为用最优法计算  $M_{ij}$  的消耗,  $kay(i, j)$  为用最优法计算  $M_{ij}$  的最后一步  $M_{ik} \times M_{kj}$  的消耗。因此  $M_{ij}$  的最优算法包括如何用最优算法计算  $M_{ik}$  和  $M_{kj}$  以及计算  $M_{ik} \times M_{kj}$ 。根据最优原理, 可得到如下的动态规划递归式:

$$\begin{aligned} c(i, i) &= 0, 1 \leq i \leq q \\ c(i, i+1) &= r_i r_{i+1} r_{i+2}; kay(i, i+1) = i, 1 \leq i \leq q \\ c(i, i+s) &= \min_{i \leq k \leq i+s} \{c(i, k) + c(k+1, i+s) + r_i r_{i+1} r_{i+s+1}\}; \\ kay(i, i+s) &= \text{获得上述最小值的 } k \\ 1 \leq i \leq q-s, 1 \leq s < q \end{aligned}$$

以上求  $c$  的递归式可用递归或迭代的方法来求解。 $c(1, q)$  为用最优法计算矩阵链的消耗,  $kay(1, q)$  为最后一步的消耗。其余的乘积可由  $kay$  值来确定。

#### (1) 递归方法

与求解 0-1 背包及图像压缩问题一样, 本递归方法也应避免重复计算  $c(i, j)$  和  $kay(i, j)$ , 否则算法的复杂度将会非常高。

**【例 13-7】** 设  $q=5$  和  $r=(10, 5, 1, 10, 2, 10)$ , 由动态规划的递归式得

$$\begin{aligned} c(1, 5) &= \min\{c(1, 1) + c(2, 5) + 500, c(1, 2) + c(3, 5) + 100, c(1, 3) + c(4, 5) + 1000, \\ &c(1, 4) + c(5, 5) + 200\} \end{aligned} \quad (13-4)$$

式中, 待求的  $c$  中有 4 个  $c$  的  $s=0$  或 1, 因此用动态规划方法可立即求得它们的值:

$$c(1, 1) = c(5, 5) = 0; c(1, 2) = 50; c(4, 5) = 200$$

下面计算  $c(2, 5)$ :

$$c(2, 5) = \min\{c(2, 2) + c(3, 5) + 50, c(2, 3) + c(4, 5) + 500, c(2, 4) + c(5, 5) + 100\} \quad (13-5)$$

式中,  $c(2, 2) = c(5, 5) = 0$ ;  $c(2, 3) = 50$ ;  $c(4, 5) = 200$ 。

再用递归式计算  $c(3, 5)$  及  $c(2, 4)$ :

$c(3, 5) = \min\{c(3, 3)+c(4, 5)+100, c(3, 4)+c(5, 5)+20\} = \min\{0+200+100, 20+0+20\} = 40$ ,  
 $c(2, 4) = \min\{c(2, 2)+c(3, 4)+10, c(2, 3)+c(4, 4)+100\} = \min\{0+20+10, 50+10+20\} = 30$

由以上计算还可得  $kay(3, 5) = 4$ ,  $kay(2, 4) = 2$ 。现在, 计算  $c(2, 5)$  所需的所有中间值都已求得, 将它们代入式 (13-5) 得  $c(2, 5) = \min\{0+40+50, 50+200+500, 30+0+100\} = 90$  且  $kay(2, 5) = 2$ ; 再用式 (13-4) 计算  $c(1, 5)$ , 在此之前必须算出  $c(3, 5)$ ,  $c(1, 3)$  和  $c(1, 4)$ 。同上述过程, 亦可计算出它们的值分别为 40、150 和 90, 相应的  $kay$  值分别为 4、2 和 2。代入式 (13-4) 得  $c(1, 5) = \min\{0+90+500, 50+40+100, 150+200+1000, 90+0+200\} = 190$  且  $kay(1, 5) = 2$ 。

此最优乘法算法的时间消耗值为 190, 由  $kay(1, 5)$  值可推出该算法的最后一步,  $kay(1, 5)=2$ , 因此最后一步为  $M_{12} \times M_{35}$ , 而  $M_{12}$  和  $M_{35}$  都是用最优法计算而来的。由  $kay(1, 2)=1$  可知  $M_{12}=M_{11} \times M_{22}$ , 同理, 由  $kay(3, 5) = 4$  可知  $M_{35}=M_{34} \times M_{55}$ 。其余类推,  $M_{34}=M_{33} \times M_{44}$ 。因而, 此最优乘法算法的步骤为:  $M_{11} \times M_{22}=M_{12}$ ,  $M_{33} \times M_{44}=M_{34}$ ,  $M_{34} \times M_{55}=M_{35}$ ,  $M_{12} \times M_{35}=M_{15}$ 。计算  $c(i, j)$  和  $kay(i, j)$  的递归算法程序代码见程序 13-6。

程序 13-6 递归计算  $c(i, j)$  和  $kay(i, j)$

---

```
int *r; //使用 r[1..n-1], 0 单元未使用, 使用前需要初始化
int **kay1; //使用 kay1[1..n-1] 单元, 0 单元未使用, 使用前需要初始化
int C(int i, int j)
{ //返回 c(i, j) 且计算 k(i, j)=kay[i][j]
    if (i==j) return 0; //一个矩阵的情形
    if (i==j-1) { //两个矩阵的情形
        kay1[i][i+1]=i; return r[i]*r[i+1]*r[i+2]; }
    //多于两个矩阵的情形, 设 u 为 k=i 时的最小值
    int u=C(i, i)+C(i+1, j)+r[i]*r[i+1]*r[j+1]; kay1[i][j]=i;
    //计算其余的最小值并更新 u
    for (int k=i+1; k<j; k++) {
        int t=C(i, k)+C(k+1, j)+r[i]*r[k+1]*r[j+1];
        if (t<u) { u=t; kay1[i][j]=k; } //小于最小值的情形
    }
    return u;
}

void Traceback (int i, int j, int **kay)
{ //输出计算  $M_{ij}$  的最优方法
    if (i==j) return;
    Traceback(i, kay[i][j], kay); Traceback(kay[i][j]+1, j, kay);
    cout << "Multiply M" << i << ", " << kay[i][j];
    cout << " and M " << (kay[i][j]+1) << ", " << j << endl;
}
```

---

函数 C 返回  $c(i, j)$  之值且置  $\text{kay}[a][b] = \text{kay}(a, b)$ , 其中  $c(a, b)$  在计算  $c(i, j)$  时已算出。函数 Traceback 利用函数 C 中已算出的  $\text{kay}$  值来推导出最优乘法算法的步骤。

设  $t(q)$  为函数 C 的复杂度, 其中  $q = j - i + 1$  (即  $M_{ij}$  是  $q$  个矩阵运算的结果)。当  $q$  为 1 或 2 时,  $t(q) = d$ , 其中  $d$  为一个常数; 而  $q > 2$  时,  $t(q) = 2 \sum_{k=1}^{q-1} t(k) + eq$ , 其中  $e$  是一个常量。

因此, 当  $q > 2$  时,  $t(q) > 2t(q-1) + e$ , 所以  $t(q) = O(2^q)$ 。函数 Traceback 的复杂度为  $O(q)$ 。

### (2) 无重复计算的递归方法

如果避免再次计算前面已经计算过的  $c$  及相应的  $\text{kay}$ , 可将复杂度降低到  $O(q^3)$ , 此时用一个全局数组  $c[i][j]$  来存储  $c(i, j)$  的值, 该数组初始值为 0。无重复计算的函数 C 的代码实现见程序 13-7。

程序 13-7 无重复计算的函数 C 的代码实现

```
int **c; //使用前需要初始化
int **kay; //使用前需要初始化
int C1(int i, int j)
{ //返回 c(i, j) 并计算 kay(i, j)=kayl[i][j]
    //避免重复计算, 检查是否已计算过
    if (c[i][j] > 0) return c[i][j];
    //若未计算, 则进行计算
    if (i==j) return 0; //一个矩阵的情形
    //两个矩阵的情形
    if (i==j-1) { kayl[i][i+1]=i; c[i][j]=r[i]*r[i+1]*r[i+2]; return c[i][j]; }
    //多于两个矩阵的情形, 设 u 为 k=i 时的最小值
    int u=C1(i, i)+C1(i+1, j)+r[i]*r[i+1]*r[j+1]; kayl[i][j]=i;
    //计算其余的最小值并更新 u
    for (int k=i+1; k<j; k++) {
        int t=C1(i, k)+C1(k+1, j)+r[i]*r[k+1]*r[j+1];
        if (t<u) { u=t; kayl[i][j]=k; } //比最小值还小
    }
    c[i][j]=u;
    return u;
}
```

为分析改进后函数 C 的复杂度, 再次使用分期计算方法。由于调用  $C(1, q)$  时每个  $c(i, j)$  ( $1 \leq i \leq j \leq q$ ) 仅被计算一次, 为计算尚未计算过的  $c(a, b)$ , 需附加的工作量  $s=j-i>1$ , 将  $s$  计入第一次计算  $c(a, b)$  时的工作量中。在依次计算  $c(a, b)$  时, 这个  $s$  会转计到每个  $c(a, b)$  的第一次计算时间  $c$  中, 因此每个  $c(i, i)$  均被计入  $s$  中。对于每个  $s$ , 要计算  $q-s+1$  个  $c(i, j)$ , 因此总的消耗为  $\sum_{s=1}^{q-1} (q-s+1) = O(q^3)$ 。

### (3) 迭代方法

$c$  的动态规划递归式可用迭代的方法来求解。若按  $s=2, 3, \dots, q-1$  的顺序计算  $c(i, i+s)$ , 则每个  $c$  和  $\text{kay}$  仅需计算一次。

【例 13-8】 考察例 13-7 中 5 个矩阵连乘的情况。

先初始化  $c(i, i)$  ( $0 \leq i \leq 5$ ) 为 0, 然后对于  $i=1, \dots, 4$ , 分别计算  $c(i, i+1)$ 。  $c(1, 2)=r_1r_2r_3=50$ ,  $c(2, 3)=50$ ,  $c(3, 4)=20$  和  $c(4, 5)=200$ 。相应的  $kay$  值分别为 1, 2, 3 和 4。当  $s=2$  时, 可得  $c(1, 3)=\min\{c(1, 1)+c(2, 3)+r_1r_2r_4, c(1, 2)+c(3, 3)+r_1r_3r_4\}=\min\{0+50+500, 50+0+100\}=150$  且  $kay(1, 3)=2$ 。用相同方法可求得  $c(2, 4)$  和  $c(3, 5)$  分别为 30 和 40, 相应  $kay$  值分别为 2 和 3。当  $s=3$  时, 需计算  $c(1, 4)$  和  $c(2, 5)$ 。计算  $c(2, 5)$  所需要的所有中间值均已知, 见式 (13-5), 代入计算公式后可得  $c(2, 5)=90$ ,  $kay(2, 5)=2$ 。  $c(1, 4)$  可用同样的公式计算。最后, 当  $s=4$  时, 可直接用式 (13-4) 式来计算  $c(1, 5)$ , 因为该式右边所有项都已知。计算  $c$  和  $kay$  的迭代程序见函数 `MatrixChain` (见程序 13-8), 该函数的复杂度为  $O(q^3)$ 。计算出  $kay$  后, 同样可用程序 13-6 中的 `Traceback` 函数推算出相应的最优乘法计算过程。

程序 13-8  $c$  和  $kay$  的迭代计算

```
void MatrixChain(int r[], int q, int **c, int **kay)
{
    //为所有的  $M_i$  计算耗费和  $kay$ 
    //初始化  $c[i][i]$ ,  $c[i][i+1]$  和  $kay[i][i+1]$ 
    for (int i=1; i<q; i++) {
        c[i][i]=0; c[i][i+1]=r[i]*r[i+1]*r[i+2]; kay[i][i+1]=i;
    }
    c[q][q]=0;
    //计算余下的  $c$  和  $kay$ 
    for (int s=2; s<q; s++)
        for (int i=1; i<=q-s; i++) {
            //k=i 时的最小项
            c[i][i+s]=c[i][i]+c[i+1][i+s]+r[i]*r[i+1]*r[i+s+1]; kay[i][i+s]=i;
            //余下的最小项
            for (int k=i+1; k<i+s; k++) {
                int t=c[i][k]+c[k+1][i+s]+r[i]*r[k+1]*r[i+s+1];
                if (t<c[i][i+s]) { c[i][i+s]=t; kay[i][i+s]=k; } //更小的最小项
            }
        }
}
```

## 习题 13

- 13-1 修改程序 13-1, 使它能同时计算出导致最优装载的  $x_i$  值。
- 13-2 修改程序 13-1, 使用一个表格来确定  $f(i, y)$  是否已被计算过。在求  $f(i, y)$  时, 若表中已经存在该值, 则直接取用; 若不存在该值, 则采用一个递归调用来计算该值。
- 13-3 编写函数 `Traceback` (见程序 13-3) 的迭代版本, 试说明两个版本的优缺点。
- 13-4 编写变长图像压缩过程中①和②的实现代码。
- 13-5 设  $G$  为有  $n$  个顶点的有向无环图,  $G$  中各顶点的编号为 1 到  $n$ , 且当  $\langle i, j \rangle$  为  $G$  中的一条边时,

有  $i < j$ 。设  $l(i, j)$  为边  $\langle i, j \rangle$  的长度：(1) 用动态规划方法计算图  $G$  中最长路径的长度，算法的时间耗费应为  $O(h+e)$ ，其中  $e$  为  $G$  中的边数；(2) 编写一个函数，利用 (1) 中所得到的结果来构造最长路径，其复杂性应为  $O(p)$ ，其中  $p$  为该路径的顶点数。

13-6 由 11.2.3 节可知，一个工程可分解为多个任务且这些任务可按拓扑顺序来完成。设任务从 1 到  $n$  编号，首先完成任务 1，然后完成任务 2，其余类推。假设有两种方法来完成任务。  $C_{i,1}$  为使用第一种方法完成任务  $i$  时的代价，  $C_{i,2}$  为使用第 2 种方法完成任务  $i$  时的代价。令  $T_{i,1}$  为第一种方法中任务  $i$  的时间耗费，  $T_{i,2}$  为第二种方法中任务  $i$  的时间耗费，并设各个  $T$  为整数。设计一个动态规划算法，以得到在时间  $t$  内完成所有任务的最小代价的方法。假定工程的代价为各任务的代价之和，工程所需的时间是各任务时间耗费之和（提示：可设  $\text{cost}(i, j)$  为在  $j$  时间内完成任务  $i$  到  $n$  的最小代价）。算法的复杂度是多少？

13-7 串  $s$  为串  $a$  中去掉某些字符而得到的子串。例如，串 “onion” 为串 “recognition” 的子串。当且仅当串  $s$  既是  $a$  的子串又是  $b$  的子串时，串  $s$  是串  $a$  和串  $b$  的公共子串。串  $s$  的长度指其所包含的字符数。试用动态规划算法得到串  $a$  和串  $b$  的最长公共子串（提示：设  $a=a_1a_2\cdots a_n$ ，  $b=b_1b_2\cdots b_m$ 。定义  $l(i, j)$  为串  $a_i\cdots a_n$  和  $b_j\cdots b_m$  最长公共子串的长度）。算法的复杂度是多少？

13-8 在串编辑问题中，给出两个串  $a = a_1a_2\cdots a_n$  和  $b = b_1b_2\cdots b_m$  及 3 个耗费函数  $C, D$  和  $I$ 。其中  $C(i, j)$  为将  $a_i$  改为  $b_j$  的耗费，  $D(i)$  为从  $a$  中删除  $a_i$  的耗费，  $I(i)$  为将  $b_i$  插入  $a$  中的耗费。通过修改、删除和插入操作可把串  $a$  改为串  $b$ 。例如，可删除所有  $a_i$ ，然后插入所有  $b_i$ ；或者当  $n \geq m$  时，可先把  $a_i$  变成  $b_i$  ( $1 \leq i \leq n$ )，然后删除其余的  $a_i$ 。整个操作序列的耗费为各个操作的耗费之和。设计一个动态规划算法来确定一个具有最少耗费的编辑操作序列（提示：定义  $c(i, j)$  为将  $a_1a_2\cdots a_i$  转变为  $b_1b_2\cdots b_j$  的最少耗费）。算法的复杂度是多少？

# 第 14 章 回 溯

一种可靠的问题求解方法是对其所有的候选解进行逐一检查，以此来获得所需要的解。但当一个问题的候选解数量非常大（如指数级）时，上述方法不太适用，因为即便使用最快的计算机也只能解决规模很小的问题。对候选解进行系统检查的方法有多种，其中回溯法和分枝定界法是较常用的两种方法，它们能够避免对很大的候选解集合进行检查，同时也能够保证在算法运行结束时找到所需要的解，因此它被称为“通用解题法”。本章主要讨论回溯方法，这种方法被用来设计货箱装船、背包、最大完备子图等问题的求解算法。

## 14.1 算 法 思 想

回溯是一个既带有系统性又带有跳跃性的搜索法。其基本思想是，按照深度优先策略从根结点出发搜索解空间树，当搜索到任一结点时，先判断该结点是否包含问题的解，如果不包含，则跳过以该结点为根的子树的搜索，逐层向根结点回溯；否则进入该子树，继续按深度优先策略进行搜索。如果要找到问题的所有解，则必须回溯到根，并搜索完根结点的所有子树才结束，而如果只求解问题的一个解，则只要搜索到问题的一个解便可结束。

此回溯过程也可通过结点扩展的方式进行描述：在一个问题的解空间中从根结点开始，开始结点既是一个活结点又是一个扩展结点，如果能从当前的扩展移动到一个新结点，那么这个新结点将变成一个活结点和新的扩展，旧的扩展仍是一个活结点；如果不能移动到一个新结点，当前的扩展就“死”了（即不再是一个活结点），那么便只能返回到最近被考察的活结点（回溯），这个活结点变成了新的扩展。当找到了答案或者回溯尽了所有的活结点时，搜索过程结束。

为了实现回溯，首先需要为问题定义一个解空间，这个空间必须至少包含问题的一个解（可能是最优的）；然后组织解空间以便它能被容易地搜索，典型的组织方法是图或树；定义了解空间的组织方法后，按照深度优先的方法在解空间中从开始结点（根结点）进行搜索，搜索到任一个结点后，如果能从该结点移动到一个新结点，则这个新结点将变成一个活结点并继续搜索，否则便只能返回到最近被考察的活结点（回溯）。当找到需要的解或者回溯了所有的活结点时，搜索过程结束。

**【例 14-1】** 设有如下背包问题： $n=3$ ,  $w=[20, 15, 15]$ ,  $p=[40, 25, 25]$ 且  $c=30$ ，采用回溯方法进行求解的过程如下。

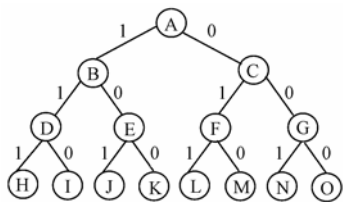


图 14.1 三个对象的背包问题的解空间

首先确定其解空间。在具有  $n$  个对象的 0-1 背包问题中，解空间可选择  $2^n$  个长度为  $n$  的 0-1 向量集合，它包含将 0 或 1 分配给  $x$  的所有可能方法。当  $n=3$  时，解空间为  $\{(0, 0, 0), (0, 1, 0), (0, 0, 1), (1, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}$ ，如图 14.1 所示。

然后从根结点开始对图 14.1 中的树进行搜

索。根结点是当前唯一的活结点，从它可移动到 B 点或 C 点，假设移动到 B，则活结点为 A 和 B。在结点 B 处，剩下的容量  $r$  为 10，收益  $cp$  为 40。从 B 点，能移动到 D 或 E。移到 D 不可行，因为所需的容量  $w_2$  为 15；到 E 的移动可行，因为没有占用任何容量，这时活结点为 A, B, E。在结点 E 处， $r=10$ ， $cp=40$ 。从 E 可移动到 J 和 K，同理，到 J 不可行而到 K 可行，由于 K 是一个叶子，所以得到一个可行的解，这个解的收益为  $cp=40$ 。 $x$  的值从根到 K 的路径来决定，路径为 A, B, E, K。从 K 向根结点进行回溯，因为 K、E 和 B 都不能进行扩展，因此这些结点都“死”了，而 A 可被进一步扩充，到达结点 C。此时  $r=30$ ， $cp=0$ 。从 C 点可移动到 F 或 G，假定移动到 F，则活结点为 A, C, F，在 F 处， $r=15$ ， $cp=25$ 。从 F 点可移动到 L 或 M，假定移动到 L，此时  $r=0$ ， $cp=50$ ，而 L 恰好是一个叶结点，它表示了一个比目前找到的最优解（即结点 K）更好的可行解，因此，可把这个解作为最优解。然后从 L 点回溯到 F 点，……，如此反复，搜索整棵树。在搜索期间发现的最优解即为最后的解。

当求解的问题需要根据  $n$  个元素的一个子集来优化某些函数时，解空间树被称作子集树。对有  $n$  个对象的 0-1 背包问题而言，它的解空间树就是一个子集树。这棵树有  $2^n$  个叶结点，全部结点有  $2^{n+1}-1$  个，因此对树中所有结点进行遍历的算法都必须耗时  $O(2^n)$ 。当求解的问题需要根据  $n$  个元素的排列来优化某些函数时，解空间树被称作排列树，这样的树有  $n!$  个叶结点，所以每一个遍历树中所有结点的算法都必须耗时  $O(n!)$ 。通过判断一个新近到达的结点能否导致一个比当前最优解还要好的解，可加速对最优解的搜索。如果不能，则移动到该结点的任何一个子树都是无意义的，可将该结点杀死。用来杀死活结点的策略称为限界函数。

回溯算法的一个特点是在搜索执行的同时产生解空间。在搜索期间的任何时刻，仅保留从开始结点到当前结点的路径，因此回溯算法的空间需求为  $O$ （从开始结点起最长路径的长度）。这个特性非常重要，因为解空间的大小通常是最长路径长度的指数或阶乘，如果要存储全部解空间的话，再多的空间也不够用。

## 14.2 应 用

### 14.2.1 货箱装船

#### (1) 问题描述

下面对前面讨论过的货箱装船问题进行一些改动。设有两艘船， $n$  个货箱，两艘船的载重量分别是  $c_1$  和  $c_2$ ， $w_i$  是货箱  $i$  的重量且  $\sum_{i=1}^n w_i \leq c_1 + c_2$ 。现在想确定是否有一种可将所有  $n$  个货箱全部装船的方法。若有的话，找出该方法。

**【例 14-2】** 当  $n = 3$ ， $c_1 = c_2 = 50$ ， $w = [10, 40, 40]$  时，可将货箱 1, 2 装到第一艘船上，货箱 3 装到第二艘船上。如果  $w = [20, 40, 40]$ ，则无法将货箱全部装船。

当  $\sum_{i=1}^n w_i = c_1 + c_2$  时，两艘船的装载问题等价于子集之和的问题，即有  $n$  个数字，要求找到一个子集使它的和为  $c_1$ 。当  $c_1 = c_2$  且  $\sum_{i=1}^n w_i = 2c_1$  时，两艘船的装载问题等价于分割问题，

即有  $n$  个数字  $a_i(1 \leq i \leq n)$ ，要求找到一个子集使得子集之和为  $\sum_{i=1}^n a_i / 2$ 。分割问题和子集之和问题都是 NP-复杂问题，所以两艘船的装载问题很难在多项式时间内解决。

当存在一种方法能够装载所有  $n$  个货箱时，可证明以下装船策略可获得成功：① 尽可能地将第一艘船装至它的重量极限；② 将剩余货箱装到第二艘船上。为了实现第①步，需要选择一个货箱的子集，它们的总重量尽可能接近  $c_1$ ，即寻找  $\max(\sum_{i=1}^n w_i a_i)$ ，其中  $\sum_{i=1}^n w_i a_i \leq c_1$ ， $a_i \in [0, 1]$ ， $0 \leq i \leq n$ 。当重量是整数时，可用动态规划方法确定第一艘船的最佳装载量，用元组方法所需时间为  $O(\min\{c_1, 2^n\})$ 。另外，也可使用回溯方法设计一个复杂度为  $O(2^n)$  的算法。在有些实例中，该方法比动态规划算法要好。

(2) 回溯算法之一

使用一个子集空间，并将其组织成如图 14.1 所示的二叉树，用深度优先方法搜索该解空间以求得最优解，并用限界函数来阻止对该结点的继续扩展搜索。如果  $Z$  是树的  $j+1$  层的一个结点，那么从根到结点  $O$  的路径定义了  $x_i (1 \leq i \leq j)$  的值，使用这些值可定义  $cw$ （当前重量）为  $\sum_{i=1}^n w_i x_i$ 。若  $cw > c_1$ ，则以结点  $O$  为根的子树不能产生一个可行解，将此判断作为限界函数，当且仅当一个结点的  $cw$  值大于  $c_1$  时，定义它是不可行的。

程序 14-1 为使用前述的限界函数所得到一个回溯算法。

程序 14-1 回溯算法之一

```
template<class T>
class Loading {
friend MaxLoading(T [ ], T, int);
private:
    void maxLoading(int i);
    int n; //货箱数目
    T *w, //货箱重量数组
    c, //第一艘船的容量
    cw, //当前装载的重量
    bestw; //目前最优装载的重量
};

template<class T>
void Loading<T>::maxLoading(int i)
{
    //从第 i 层结点开始搜索
    if (i>n) { if (cw>bestw) bestw=cw; return; } //位于叶结点
    //检查子树
    if (cw+w[i]<=c) { cw+=w[i]; maxLoading(i+1); cw-=w[i]; } //尝试 x[i]=1
    maxLoading(i+1); //尝试 x[i]=0
}

template<class T>
T MaxLoading(T w[ ], T c, int n)
{
    //返回最优装载的重量
}
```



```

Loading<T> X;
X.w=w; X.c=c; X.n=n; X.bestw=0; X.cw=0;  //初始化 X
X.maxLoading(1); //计算最优装载的重量
return X.bestw;
}

```

函数 `maxLoading` 在它到达的每一个结点上花费  $O(1)$  时间，到达的结点数量为  $O(2^n)$ ，所以复杂度为  $O(2^n)$ ，并且需要使用的递归栈空间为  $O(n)$ 。

### (3) 回溯方法之二

如果对于不可能包含比当前最优解还要好的解的右子树进行搜索，能更进一步地提高函数 `maxLoading` 的性能。令 `bestw` 为当前最优解的重量， $Z$  为解空间树的第  $i$  层的一个结点，`cw` 的定义如前。以结点  $Z$  为根的子树中没有叶结点的重量会超过 `cw+r`，其中  $r = \sum_{j=1}^n w[j]$  为剩余货箱的重量。因此，当 `cw+r ≤ bestw` 时，没有必要去搜索结点  $Z$  的右子树。此方法相当于加强了限界函数的条件，修改后的算法见程序 14-2。

#### 程序 14-2 回溯方法之二——程序 14-1 的优化

```

//class Loading 定义如前，但需新增一数据成员 int r，表示剩余货物的重量
template<class T>
void Loading<T>::maxLoading(int i)
{
    //从第 i 层结点开始搜索
    if (i>n) { bestw=cw; return; } //在叶结点上
    //检查子树
    r-=w[i];
    if (cw+w[i]<=c) { cw+=w[i]; maxLoading(i+1); cw-=w[i]; } //尝试 x[i]=1
    if (cw+r>bestw) maxLoading(i+1); r+=w[i]; //尝试 x[i]=0
}
template<class T>
T MaxLoading(T w[ ], T c, int n)
{
    //返回最优装载的重量
    Loading<T>X;
    X.w=w; X.c=c; X.n=n; X.bestw=0; X.cw=0;  //初始化 X
    X.r=0; //r 的初始值为所有重量之和
    for (int i=1; i<=n; i++) X.r+=w[i];
    //计算最优装载的重量
    X.maxLoading(1); return X.bestw;
}

```

在该算法程序中，由于加强的限界函数不允许移动到不能产生较好解的结点，因此每到达一个新的叶结点就意味着找到了比当前最优解还优的解。虽然新代码的复杂度仍是  $O(2^n)$ ，但它可比程序 14-1 要少搜索一些结点。

### (4) 寻找最优子集

为确定具有最接近重量  $c$  的货箱子集，需要记录当前找到的最优子集，可将一个整数数

组 bestx 添加到 MaxLoading 中，其中元素可为 0 或 1，当且仅当  $\text{bestx}[i] = 1$  时，货箱  $i$  在最优子集中，新的代码见程序 14-3。

程序 14-3 给出最优装载的代码

```
//class Loading 定义如前，但需新增两个数据成员 int *x, *bestx
//数组 x 用来记录从搜索树的根到当前结点的路径(即它保留了路径上的 xi 值)
//bestx 记录当前最优解 template<class T>
void Loading<T>::maxLoading(int i)
{ //从第 i 层结点开始搜索，在叶结点上
    if (i>n) { for (int j=1; j<=n; j++) bestx[j]=x[j]; bestw=cw; return; }
    //检查子树
    r-=w[i];
    if (cw+w[i]<=c) { //尝试 x[i]=1
        x[i]=1; cw+=w[i]; maxLoading(i+1); cw-=w[i]; }
    if (cw+r>bestw) { x[i]=0; maxLoading(i+1); } //尝试 x[i]=0
        r+=w[i];
    }
template<class T>
T MaxLoading(T w[ ], T c, int n, int bestx[ ])
{ //返回最优装载及其值
    Loading<T> X;
    X.x=new int [n+1]; X.w=w; X.c=c; //初始化 X
    X.n=n; X.bestx=bestx; X.bestw=0; X.cw=0; //初始化 X
    //r 的初始值为所有重量之和
    X.r=0;
    for (int i=1; i<=n; i++) X.r+=w[i]; X.maxLoading(1); delete [ ] X.x;
    return X.bestw;
}
```

### (5) 一个改进的迭代版本

可进一步改进程序 14-3 的代码以减少它的空间需求。由于数组  $x$  中记录了在树进行移动的所有路径，因此可消除大小为  $O(n)$  的递归栈空间。从解空间树的任何结点开始，算法不断向左孩子移动，直到不能再移动为止。如果一个叶子已被到达，则最优解被更新。否则，它试图移动到右孩子，结果是，要么到达一个叶结点，要么不值得移动到一个右孩子。算法回溯到一个结点的条件是，从该结点向其右孩子移动有可能找到最优解，而这个结点具有一个特性，即它是路径中具有  $x[i]=1$  的结点中离根结点最近的结点。如果向右孩子的移动是有效的，那么就进行移动，然后再完成一系列向左孩子的移动。如果向右孩子的移动是无效的，则回溯到  $x[i]=1$  的下一个结点。该算法遍历树的方式可被编码成迭代（即循环）算法，算法的具体代码实现留给作者自行实现。

## 14.2.2 0-1 背包问题

前面章节分别介绍了对 0-1 背包问题采用贪婪算法、动态规划算法的解题思路。本节将

用回溯算法解决这个问题。由于该问题的解空间可组织成子集树的形状，如图 14.1 所示，因此回溯算法与装载问题很类似。在判断是否要移动到右子树时，一个简单方法是令  $r$  为还未遍历的对象的收益之和，将  $r$  加到  $cp$ （当前结点所获收益）之上，若  $(r+cp) \leq bestp$ （当前最优解的收益），则不需搜索右子树。另一种更有效的方法是按收益密度  $p_i/w_i$  对剩余对象进行排序，将对象按密度递减的顺序去填充背包的剩余容量，当遇到第一个不能全部放入背包的对象时，就使用它的一部分。

**【例 14-3】** 考察一个背包例子： $n=4$ ， $c=7$ ， $p=[9, 10, 7, 4]$ ， $w=[3, 5, 2, 1]$ ，其收益密度为 $[3, 2, 3.5, 4]$ 。当背包以密度递减的顺序被填充时，对象 4 首先被填充，然后是 3 和 1，当这 3 个对象被装入背包之后，剩余容量为 1。这个容量可容纳对象 2 的 0.2 倍的重量，将 0.2 倍的该对象装入，产生了收益值 2。被构造的解为  $x=[1, 0.2, 1, 1]$ ，相应的收益值为 22。尽管该解不可行（ $x_2$  是 0.2，而实际上它应为 0 或 1），但它的收益值 22 一定不少于要求的最优解。因此，该 0-1 背包问题没有收益值多于 22 的解。

程序 14-4 所示的计算最优解收益值的类 **Knap** 定义中，采用了将对象按密度进行排序的限界函数实现方法，这样能减少限界函数 **Bound**（见程序 14-5）及递归函数 **Knapsack**（见程序 14-6）的参数数量，同时也减少了递归栈空间以及每一个 **Knapsack** 的执行时间。该函数与 **maxLoading**（见程序 14-2）类似。注意，该程序中的限界函数仅在向右孩子移动时才计算，而向左孩子移动时，左孩子的限界函数的值与其父结点相同。

程序 14-4 Knap 类

```
template<class Tw, class Tp>
class Knap {
friend Tp Knapsack(Tp *, Tw *, Tw, int);
private:
    Tp Bound(int i);
    void Knapsack(int i);
    Tw c; //背包容量
    int n; //对象数目
    Tw *w; //对象重量的数组
    Tp *p; //对象收益的数组
    Tw cw; //当前背包的重量
    Tp cp; //当前背包的收益
    Tp bestp; //迄今最大的收益
};
```

程序 14-5 限界函数

```
template<class Tw, class Tp>
Tp Knap<Tw, Tp>:: Bound(int i)
{//返回子树中最优叶子的上限值 Return upper bound on value of
//best leaf in subtree.
    Tw cleft=c-cw; //剩余容量
    Tp b=cp; //收益的界限
    //按照收益密度的次序装填剩余容量
```

```

while (i<=n && w[i]<=cleft) { cleft-=w[i]; b+=p[i]; i++; }
if (i<=n) b+=p[i]/w[i] * cleft; //取下一个对象的一部分
return b;
}

```

---

### 程序 14-6 0-1 背包问题的递归函数

---

```

template<class Tw, class Tp>
void Knap<Tw, Tp>:: Knapsack(int i)
{//从第 i 层结点搜索
    if (i>n) { bestp=cp; return; } //在叶结点上
    //检查子树
    if (cw+w[i]<=c) { //尝试 x[i]=1
        cw+=w[i]; cp+=p[i]; Knapsack(i+1); cw-=w[i]; cp-=p[i]; }
    if (Bound(i+1)> bestp) Knapsack(i+1); //尝试 x[i]=0
}

```

程序 14-7 定义了类 Object 来实现按密度对对象进行排序。

---

### 程序 14-7 Object 类

---

```

class Object {
friend int Knapsack(int *, int *, int, int);
public:
    int operator<=(Object a) const
    { return (d>=a.d); }
private:
    int ID; //对象号
    float d; //收益密度
};

```

程序 14-8 中 Knapsack 的复杂度为  $O(n2^n)$ ，这是因为，限界函数的复杂度为  $O(n)$ ，且该函数在  $2^n$  个右孩子处被计算。

---

### 程序 14-8 程序 14-6 的预处理代码

---

```

template<class Tw, class Tp>
Tp Knapsack(Tp p[ ], Tw w[ ], Tw c, int n)
{//返回最优装包的值
    //初始化
    Tw W=0; //记录重量之和
    Tp P=0; //记录收益之和
    //定义一个按收益密度排序的对象数组
    Object *Q=new Object [n];
    for (int i=1; i<=n; i++) {
        //收益密度的数组
        Q[i-1].ID=i; Q[i-1].d=1.0*p[i]/w[i]; //注意到 p[0], w[0]不使用
    }
}

```

```

        P+=p[i]; W+=w[i];
    }
    if (W<=c) return P; //可容纳所有对象
    MergeSort(Q, n); //按密度排序
    //创建 K n a p 的成员
    Knap<Tw, Tp> K; K.p=new Tp [n+1]; K.w=new Tw [n+1];
    for (i=1; i<=n; i++) { K.p[i]=p[Q[i-1].ID]; K.w[i]=w[Q[i-1].ID]; }
    K.cp=0; K.cw=0; K.c=c; K.n=n; K.bestp=0;
    //寻找最优收益
    K.Knapsack(1);
    delete [ ] Q; delete [ ] K.w; delete [ ] K.p;
    return K.bestp;
}

```

### 14.2.3 最大完备子图

对于一个无向图  $G$ , 设  $U$  为顶点的子集, 当且仅当  $U$  中的任意点  $u$  和  $v$ , 在图  $G$  中存在一条  $(u, v)$  边时,  $U$  为一个完全子图, 子图的大小为图中顶点的数量。当且仅当一个完全子图不被包含在  $G$  的一个更大的完全子图中时, 它是图  $G$  的一个完备子图。最大完备子图问题是寻找图  $G$  中具有最大顶点数的最大完备子图。

**【例 14-4】** 在图 14.2 (a) 中, 子集  $\{2, 3\}$  定义了一个大小为 2 的完全子图, 但它不是一个完备子图, 因为它被包含在一个更大的完全子图  $\{1, 2, 4\}$  中。 $\{1, 2, 4\}$  定义了该图的一个最大的完备子图。点集  $\{2, 3, 4\}$  定义了另一个最大的完备子图。

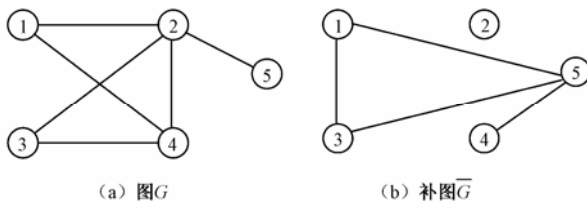


图 14.2 图及其补图

当且仅当对于  $U$  中任意点  $u$  和  $v$ ,  $(u, v)$  不是  $G$  的一条边时,  $U$  定义了一个空子图; 当且仅当一个子集不被包含在一个更大的点集中时, 该点集是图  $G$  的一个独立集, 同时它也定义了图  $G$  的空子图。最大独立集是具有最大规模的独立集。

任意图  $G$ , 它的补图  $\bar{G}$  是有同样点集的图, 且当且仅当  $(u, v)$  不是  $G$  的一条边时, 它是  $\bar{G}$  的一条边。

**【例 14-5】** 图 14.2 (b) 是图 14.2 (a) 的补图, 反之亦然。 $\{1, 3, 5\}$  定义了 14.2 (a) 的一个空子图, 它也是该图的一个最大独立集。虽然  $\{1, 2\}$  定义了图 14.2 (b) 的一个空子图, 但它不是一个独立集, 因为它被包含在空子图  $\{1, 2, 4\}$  中。 $\{1, 2, 4\}$  是图 14.2 (b) 中的一个最大独立集。

如果  $U$  定义了  $G$  的一个完全子图, 则它也定义了  $\bar{G}$  的一个空子图, 反之亦然。因此  $G$  的完备子图与  $\bar{G}$  的独立集之间有对应关系, 特别是,  $G$  的一个最大完备子图定义了  $\bar{G}$  的一

个最大独立集。

类似于最大完备子图问题，最大独立集问题是指寻找图  $G$  的一个最大独立集。这两个问题都是 NP-复杂问题，一个问题的解决也就意味着同时解决了另一个问题。例如，有一个算法能求解最大完备子图问题，则它同时也能求解最大独立集问题。实现的途径是，首先求出所给图的补图，然后寻找补图的最大完备子图。

求解最大独立集问题可看成是寻找一个具有最大规模的互不交叉的网组集合问题。给定一个图，图中每个顶点表示一个网组，当且仅当两个顶点对应的网组交叉时，它们之间有一条边，因此该图的一个最大独立集为非交叉网组的一个最大规模的子集。若网组有一个端点在路径顶端，另一个在底端时，非交叉网组的最大规模子集能在多项式时间（实际上是  $O(n^2)$ ）内用动态规划算法得到。但当一个网组的端点可能在平面中的任意地方时，不可能有在多项式时间内找到非交叉网组的最大规模子集的算法。

最大完备子图问题和最大独立集问题可由回溯算法在  $O(n2^n)$  时间内解决，两者都可使用如图 14.1 所示的子集解空间树。最大完备子图问题的递归回溯算法与程序 14-3 非常类似，即在任意时刻，当要对解空间树的第  $i$  层结点  $Z$  的左孩子进行访问时，需要证明从顶点  $i$  到每一个其他的顶点  $j$  ( $x_j = 1$  且  $j$  在从根到  $Z$  的路径上) 有一条边；当要移动到  $Z$  的右孩子时，需要证明还有足够多的顶点未被搜索，以便在右子树有可能找到一个较大的完备子图。最大完备子图算法实现见程序 14-9。

#### 程序 14-9 最大完备子图

```
//类中补充定义:
protected:
    int *x, //存储当前结点的路径
    *bestx, //保存目前的最优解
    bestn, //bestx 中点的数量
    cn; //x 中点的数量
    void maxClique(int i);
public:
    int MaxClique(int v[ ]);

template<class VertexType, class EdgeType>
void GraphMatrixND<VertexType, EdgeType>::maxClique(int i)
{//计算最大完备子图的回溯代码
    if (i>numNodes) { //在叶子上找到一个更大的完备子图，更新
        for (int j=1; j<=numNodes; j++) bestx[j]=x[j];
        bestn=cn; return; } //if
    //在当前完备子图中检查顶点 i 是否与其他顶点相连
    int OK=1;
    for (int j=1; j<i; j++)
        if (x[j] && smpEdges[i-1][j-1]==0) { OK=0; break; } //i 不与 j 相连
    if (OK) { //尝试 x[i]=1, 把 i 加入完备子图
        x[i]=1; cn++; maxClique(i+1); x[i]=0; cn--; } //if
    if (cn+numNodes-i>bestn) { x[i]=0; maxClique(i+1); } //尝试 x[i]=0
```

```

}

template<class VertexType, class EdgeType>
int GraphMatrixND<VertexType, EdgeType>::MaxClique(int v[ ])
{
    //返回最大完备子图的大小
    //完备子图的顶点放入 v [ 1:n ]
    x=new int [numNodes+1]; cn=0; bestn=0; bestx=v; //初始化
    maxClique(1); delete [ ] x; //寻找最大完备子图
    return bestn;
}

```

在上述程序中，回溯算法作为类 GraphMatrixND 的一个成员来实现，函数 MaxClique 对解空间树进行搜索，而 MaxClique(1)初始化必要的变量。MaxClique(v)的执行返回最大完备子图的尺寸。

## 习题 14

14-1 考察如下 0-1 背包问题： $n = 4$ ,  $w = [20, 25, 15, 35]$ ,  $p = [40, 49, 25, 60]$ ,  $c = 62$ 。(1) 画出该 0-1 背包问题的解空间树；(2) 对该树运用回溯算法（利用给出的  $p_s$ ,  $w_s$ ,  $c$  值），依回溯算法遍历结点的顺序标记结点，并确定回溯算法未遍历的结点。

14-2 用迭代回溯算法求解 0-1 背包问题。

14-3 令  $G$  为一个无向图。当且仅当对于  $G$  中的每一条边  $(u, v)$ ,  $u$  或  $v$  或  $u, v$  在  $U$  中时,  $G$  的顶点子集  $U$  是一个顶点覆盖。 $U$  中顶点的数量是覆盖的大小。在图 14.2 (a) 中,  $\{1, 2, 4\}$  是大小为 3 的一个顶点覆盖。编写一个回溯算法寻找具有最小尺寸的顶点覆盖。问算法的复杂度是多少?

14-4 令  $G$  是一个无向图,  $U$  是  $G$  中顶点的任意子集,  $V$  是  $G$  余下的点的集合。一个端点在  $U$  中, 另一个端点在  $V$  中的边的数量是  $U$  所定义的切割的大小。编写一个回溯算法, 寻找最大切割的大小和相应的  $U$ 。问算法的复杂度是多少?

14-5 编写一个函数, 用回溯法搜索一个解空间。函数中的参数应包括下列函数: 产生结点的下一个孩子的函数, 决定下一个孩子是否是可行的函数, 计算该结点界限的函数, 决定该界限值是否优于另一个值的函数等。用 0-1 背包问题来测试程序。

## 第 15 章 分枝定界法

分枝定界法和前面介绍的回溯法相似，也是在解空间中求出问题的解，但不同的是，分枝定界法一般采用广度优先或最小耗费方法来搜索解空间的树结构。本章通过采用与第 14 章相同的应用例子来进行讨论，以便更容易地比较分枝定界法与回溯法的异同。相对而言，分枝定界法的解空间比回溯法的大得多，因此当内存容量有限时，回溯法成功的可能性更大。

### 15.1 算 法 思 想

分枝定界法是另一种系统地搜索解空间的方法，其解空间树也为一棵有序树，如排序树或子集树。它与回溯法的主要区别在于对当前扩展结点所采用的扩展方式不同。在分枝定界法中，每一个活结点有且仅有一次机会变成扩展结点，活结点一旦变为扩展结点，就会一次性地生成其所有的孩子结点（即新结点），在这些孩子结点中，那些不可能导出可行解或最优解的结点将被舍弃，其余的孩子结点被加入活结点表中。然后从表中选择一个结点作为下一个扩展结点，并重复上述结点的扩展过程，直到找到所需要的解或活动表为空时结束。

从活结点表中选择下一个扩展结点通常有两种方式。①先进先出（FIFO）法。即从活结点表中取出结点的顺序与加入结点的顺序相同，因此活结点表的性质与队列的相同。②最小耗费或最大收益法。由于解空间中每个结点都有一个对应的耗费或收益，可将活结点表组织成一个优先队列，并根据结点的耗费所确定的结点优先级别来选取下一个结点。如果查找的是具有最小耗费的解，则活结点表可用最小堆来建立，下一个扩展结点就是具有最小耗费的活结点；如果查找的是具有最大收益的解，则可用最大堆来构造活结点表，下一个扩展结点便是具有最大收益的活结点。

**【例 15-1】** 下面分别用 FIFO 分枝定界法和最大收益分枝定界方法解决例 14-1 的背包问题并进行比较，即  $n = 3$ ,  $w = [20, 15, 15]$ ,  $p = [40, 25, 25]$ ,  $c = 30$ ，它们的解空间也与例 14-1 的解空间相同。

① FIFO 分枝定界利用一个队列来记录活结点，结点按照 FIFO 顺序从队列中取出。在该方法的搜索过程中，初始以根结点 A 作为扩展结点，活结点队列为空，对 A 进行扩展时，生成结点 B 和 C，由于这两个结点都是可行的，因此都被加入活结点队列中，结点 A 被删除。下一个扩展结点 B，产生结点 D 和 E，由于 D 是不可行的，因此被删除，而 E 被加入队列中。下一步选择结点 C 为扩展结点，生成结点 F 和 G，两者都是可行结点，加入队列中。下一个扩展结点 E 生成结点 J 和 K，J 不可行而被删除，K 是一个可行的叶结点，并产生一个到目前为止可行的解，它的收益值为 40。

接下来扩展结点 F，它产生两个孩子 L 和 M，L 代表一个可行的解且其收益值为 50，M 代表另一个收益值为 15 的可行解。G 是最后一个扩展结点，它的孩子 N 和 O 都是可行的。由于活结点队列变为空，因此搜索过程终止，最佳解的收益值为 50。

从上述过程可看出，在解空间树上进行的 FIFO 分枝定界方法类似于从根结点出发的广度优先搜索，它们的主要区别是在 FIFO 分枝定界法中不可行的结点不会被搜索。



② 最大收益分枝定界法使用一个最大堆，其中的扩展结点按照每个活结点收益值的降序，或者按照活结点任意子树的叶结点所能获得的收益估计值的降序，从队列中取出。该方法的搜索过程也以解空间树中的结点 A 作为初始结点，进行扩展后得到结点 B 和 C，两者都是可行的并被加入堆中，结点 B 获得的收益值是 40（设  $x_1=1$ ），C 获得的收益值为 0。A 被删除，由于 B 的收益值比 C 的大，因此对 B 进行扩展，得到结点 D 和 E，D 是不可行的而被删除，E 加入堆中。由于 E 具有收益值 40，而 C 为 0，因为 E 成为下一个扩展结点，生成结点 J 和 K，J 不可行而被删除，K 是一个可行的解，因此 K 作为目前能找到的最优解而被记录下来，然后 K 被删除。由于只剩下一个活结点 C 在堆中，因此 C 作为扩展结点被展开，生成 F、G 两个结点加入堆中。F 的收益值为 25，因此成为下一个扩展结点，展开后得到结点 L 和 M，由于 L 和 M 是叶结点，因此都被删除，同时 L 所对应的解作为当前最优解被记录下来。最后 G 成为扩展结点，生成的结点为 N 和 O，两者都是叶结点而被删除，两者所对应的解都不比当前的最优解更好，因此最优解保持不变，此时堆变为空，搜索过程结束，到达结点 J 的搜索为最优解。

与回溯法类似，也可使用一个定界函数来减少所产生的解空间树结点数目，以此加速最优解的搜索过程。定界函数为最大收益设置了一个上限，通过展开一个特殊的结点可能获得这个最大收益。如果一个结点的定界函数值不大于目前最优解的收益值，则此结点被删除而不展开。更进一步地，在最大收益分枝定界法中，可使结点按照它们收益的定界函数值的非升序从堆中取出，使搜索过程从可能到达一个好的叶结点的活结点出发，而不是从目前具有较大收益值的结点出发。

回溯法比分枝定界法占用更少的内存空间，回溯法占用的内存是  $O(\text{解空间的最大路径长度})$ ，而分枝定界所占用的内存为  $O(\text{解空间大小})$ 。对于一个子集空间，回溯法需要  $O(n)$  的内存空间，而分枝定界则需要  $O(2^n)$  的空间。对于排列空间，回溯需要  $O(n)$  的内存空间，分枝定界需要  $O(n!)$  的空间。虽然最大收益（或最小耗费）分枝定界法在许多情况下可能会比回溯法检查更少的结点，但在实际应用中，它可能在回溯法超出允许的时间限制之前就超出了内存的限制。

## 15.2 应 用

### 15.2.1 货箱装船

#### (1) FIFO 分枝定界法

前面 14.2.1 节的货箱装船问题主要是寻找第一条船的最大装载方案，该问题的解空间是一棵子集树。对程序 14-1 进行改造便可得到程序 15-1 所示的 FIFO 分枝定界法的程序代码，该方法只是寻找最大装载的重量。

程序 15-1 货箱装船问题的 FIFO 分枝定界法

```
#include "LinkedList.h"//类 LinkedList
template<class T>
void AddLiveNode(LinkedList<T> &Q, T wt, T& bestw, int i, int n)
{//如果不是叶结点，则将结点权值 wt 加入队列 Q
if (i==n) { if (wt>bestw) bestw=wt; } else Q.Insert(wt); }//叶子，不是叶子
```

```

template<class T>
T MaxLoading(Tw[ ], T c, int n)
{ //返回最优装载值
//使用 FIFO 分枝定界算法
//为层次 1 初始化
    LinkedList<T> Q; //活结点队列
    Q.Insert(-1); //标记本层的尾部
    int i=1; //扩展结点的层
    TEw=0; //扩展结点的权值
    bestw=0; //目前的最优值
    //搜索子集空间树
    while (true) {
        //检查扩展结点的左孩子
        if (Ew+w[i]<=c) AddLiveNode(Q, Ew+w[i], bestw, i, n); //x[i]=1
        //右孩子总是可行的
        AddLiveNode(Q, Ew, bestw, i, n); //x[i]=0
        Q.Delete(Ew); //取下一个扩展结点
        if (Ew==-1) { //到达层的尾部
            if (Q.IsEmpty()) return bestw;
            Q.Insert(-1); //添加尾部标记
            Q.Delete(Ew); //取下一个扩展结点
            i++; } //Ew 的层
    } //while
}

```

---

程序中，函数 **MaxLoading** 在解空间树中进行分枝定界搜索。链表队列 **Q** 用于保存活结点，它记录各活结点对应的权值，另外还记录了权值-1，以标识每一层活结点的结尾。函数 **AddLiveNode** 用来在搜索过程中增加结点。搜索中所到达的每个叶结点都对应着一个可行的解，而每个解都会与目前的最优解进行比较，以确定最优解。**MaxLoading** 函数的时间和空间复杂度都是  $O(2^n)$ 。

## (2) 改进

可通过使用定界函数来改进上述问题的求解过程，即只有当右孩子对应的重量加上剩余货箱的重量超出当前最优解时才选择右孩子，如程序 15-2 所示。

程序 15-2 对程序 15-1 的改进

---

```

template<class T>
T MaxLoading1(Tw[ ], Tc, int n)
{ //返回最优装载值
//使用 FIFO 分枝定界算法
//为层 1 初始化
    int r;
    LinkedList<T> Q; //活结点队列
    Q.Insert(-1); //标记本层的尾部

```

```

int i=1; //扩展结点的层
T Ew=0, //扩展结点的重量
bestw=0; //目前的最优值
r=0; //扩展结点中余下的重量
for (int j=2; j<=n; j++) r+=w[i];
//搜索子集空间树
while (true) { //检查扩展结点的左孩子
    T wt=Ew+w[i]; //左孩子的权值
    if (wt<=c) { //可行的左孩子
        if (wt>bestw) bestw=wt; //若不是叶子，则添加到队列中
        if (i<n) Q.Insert(wt); }
        if (Ew+r>bestw && i<n) //检查右孩子
            Q.Insert(Ew); //可有一个更好的叶子
    Q.Delete(Ew); //取下一个扩展结点
    if (Ew==-1) { //到达层的尾部
        if (Q.IsEmpty( )) return bestw;
        Q.Insert(-1); //添加尾部标记
        Q.Delete(Ew); //取下一个扩展结点
        i++; //扩展结点的层
        r-=w[i]; } //扩展结点中余下的重量
    }
}

```

---

### (3) 寻找最优子集

为了找到最优子集，需要记录从每个活结点到达根的路径，因此在找到最优装载所对应的叶结点之后，可利用所记录的路径返回到根结点来设置 *x* 的值。活结点队列中元素的类型是 *QNode*（见程序 15-3）。这里，当且仅当结点是它的父结点的左孩子时，*LChild* 为 *true*。程序 15-4 是计算最优子集的分枝定界法代码。

程序 15-3 *QNode* 类

---

```

template<class T>
class QNode {
friend void AddLiveNode (LinkedList<QNode<T>*> &Q, T wt, int i, int n, T bestw,
QNode<T> *E, QNode<T> *&bestE, int bestx[ ], bool ch);
friend T MaxLoading(Tw[ ], Tc, int n, int bestx[ ]);
private:
    QNode *parent; //父结点指针
    bool LChild; //当且仅当是父结点的左孩子时，取值为 true
    T weight; //由到达本结点的路径所定义的部分解的值
};

```

---

```

template<class T>
void AddLiveNode(LinkedQueue<QNode<T>*> &Q, T wt, int i, int n, T bestw,
QNode<T> *E, QNode<T> *&bestE, int bestx[ ], bool ch)
{//如果不是叶结点, 则向队列 Q 中添加一个 i 层、重量为 wt 的活结点
//新结点是 E 的一个孩子。当且仅当新结点是左孩子时, ch 为 true
//若是叶子, 则 ch 取值为 bestx[n]
    if (i==n)  {//叶子
        if (wt==bestw)  {  bestE=E; bestx[n]=ch; }  //目前的最优解
    return; }
    //不是叶子, 添加到队列中
    QNode<T> *b;
    b=new QNode<T>; b->weight=wt; b->parent=E; b->LChild=ch;
    Q.Insert(b);
}

template<class T>
T MaxLoading(Tw[ ], Tc, int n, int bestx[ ])
{//返回最优装载值, 并在 bestx 中返回最优装载方案
//使用 FIFO 分枝定界算法
//初始化层 1
    int r;
    LinkedQueue<QNode<T>*> Q;  //活结点队列
    Q.Insert(0);  //0 代表本层的尾部
    int i=1;  //扩展结点的层
    T Ew=0,  //扩展结点的重量
    bestw=0;  //迄今得到的最优值
    r=0;  //扩展结点中余下的重量
    for (int j=2; j<=n; j++)  r+=w[i];
    QNode<T> *E=0;  //当前的扩展结点
    *bestE;  //目前最优的扩展结点
    //搜索子集空间树
    while (true)  {  //检查扩展结点的左孩子
        T wt=Ew+w[i];
        if (wt<=c)  {//可行的左孩子
            if (wt>bestw)  bestw=wt;
            AddLiveNode(Q, wt, i, n, bestw, E, bestE, bestx, true); }
        //检查右孩子
        if (Ew+r>bestw)  AddLiveNode(Q, Ew, i, n, bestw, E, bestE, bestx, false);
        Q.Delete(E);  //下一个扩展结点
        if (!E)  {//层的尾部
            if (Q.IsEmpty( ))  break;
            Q.Insert(0);  //层尾指针
            Q.Delete(E);  //下一个扩展结点
            i++;  //扩展结点的层次
        }
    }
}

```

```

        r-=w[i]; } //扩展结点中余下的重量
        Ew=扩展>weight; //新的扩展结点的重量
    }
    //沿着从 bestE 到根的路径构造 x[ ], x[n]由 AddLiveNode 来设置
    for (j=n-1; j>0; j--) {
        bestx[j]=best 扩展>LChild; //从 bool 转换为 int
        bestE=best 扩展>parent;
    }
    return bestw;
}

```

---

#### (4) 最大收益分枝定界

在对子集树进行最大收益分枝定界搜索时，活结点列表是一个最大优先级队列，其中每个活结点  $x$  都有一个相应的重量上限（即最大收益）。这个重量上限是结点  $x$  相应的重量加上剩余货箱的总重量，所有的活结点按其重量上限的递减顺序变为扩展结点。需要注意的是，对任意一个结点  $x$ ，其子树中不可能存在重量超过结点  $x$  重量的结点。另外，当叶结点对应的重量等于它的重量上限时，可得出结论：在最大收益分枝定界算法中，当某个叶结点成为扩展结点并且其他任何活结点都不可能扩展到具有更大重量的叶结点时，最优装载的搜索终止。

上述策略可用两种方法来实现。一种方法是，最大优先级队列中的活结点都是互相独立的，因此每个活结点内部必须记录从子集树的根到此结点的路径。一旦找到了最优装载所对应的叶结点，就利用这些路径信息来计算  $x$  值。另一种方法是，除了把结点加入最大优先级队列之外，结点还放在另一个独立的树结构中，这个树结构用来表示所生成的子集树的一部分。当找到最大装载之后，就可沿着路径从叶结点一步一步返回到根，从而计算出  $x$  值。下面给出后一种方法的实现过程，前一种方法的实现留给读者实现。

程序 15-5 用 `HeapNode` 类型的最大堆来表示最大优先级队列，用 `bbnode` 类型来表示子集树中的结点。

程序 15-5 `bbnode` 类和 `HeapNode` 类

---

```

class bbnode {
public:
    bbnode *parent; //父结点指针
    bool LChild; //当且仅当是父结点的左孩子时，取值为 true
};

template<class T>
class HeapNode{
public:
    operator T( ) const { return uweight; }
    bbnode *ptr; //活结点指针
    T uweight; //活结点的重量上限
    int level; //活结点所在层
};

```

---

程序 15-6 中的函数 `AddLiveNode` 用于把 `bbnode` 类型的活结点加到子树中，并把 `HeapNode` 类型的活结点插入最大堆中。

#### 程序 15-6 加入活结点

```
template<class T>
void AddLiveNode2(MaxHeap<HeapNode<T>>&H, bbnode *E, T wt, bool ch, int lev)
{//向最大堆 H 中增添一个层为 lev，上限重量为 wt 的活结点
//新结点是 E 的一个孩子
//当且仅当新结点是左孩子时，ch 为 true
    bbnode *b=new bbnode; b->parent=E; b->LChild=ch;
    HeapNode<T> N;
    N.ueight=wt; N.level=lev; N.ptr=b; H.Insert(N);
}

template<class T>
T MaxLoading2(Tw[ ], Tc, int n, int bestx[ ])
{//返回最优装载值，最优装载方案保存于 bestx
//使用最大收益分枝定界算法
//定义一个最多有 1000 个活结点的最大堆
    MaxHeap<HeapNode<T>>H(1000);
    //第一剩余重量的数组
    //r[j]为 w[j+1:n]的重量之和
    T *r=new T [n+1]; r[n]=0;
    for (int j=n-1; j>0; j--) r[j]=r[j+1]+w[j+1];
    //初始化层 1
    int i=1; //扩展结点的层
    bbnode *E=0, //当前扩展结点
    T Ew=0; //扩展结点的重量
    //搜索子集空间树
    while (i!=n+1) { //不在叶子上
        //检查扩展结点的孩子
        if (Ew+w[i]<=c) { //可行的左孩子
            AddLiveNode2(H, E, Ew+w[i]+r[i], true, i+1);
        } //if
        //右孩子
        AddLiveNode2(H, E, Ew+r[i], false, i+1);
        //取下一个扩展结点
        HeapNode<T> N; H.DeleteMax(N); //不能为空
        i=N.level; E=N.ptr; Ew=N.ueight-r[i-1];
    } //while
    //沿着从扩展结点 E 到根的路径构造 bestx[ ]
    for (j=n; j>0; j--) { bestx[j]=扩展>LChild; //从 bool 转换为 int
        //E=扩展>parent;
    } //for
    return Ew;
}
```

上述程序中的函数 `MaxLoading` 定义了一个容量为 1000 的最大堆，因此，可用它来解决优先队列中活结点数在任何时候都不超过 1000 的装箱问题。

### 15.2.2 0-1 背包问题

0-1 背包问题的最大收益分枝定界算法（见程序 15-7）可通过程序 15-6 产生，它定义类 `Knap` 类似于回溯法中的类 `Knap`（见程序 14-4），函数 `MaxProfitKnapsack` 在子集树中执行最大收益分枝定界搜索。

程序 15-7 0-1 背包问题的最大收益分枝定界法

---

```
template<class Tw, class Tp>
class Knap {
public:
    friend Tp Knapsack(Tp *, Tw *, Tw, int);
    Tp MaxProfitKnapsack( );
    Knap(int c, int*w, int *p, int num);
private:
    Tp Bound(int i);
    void Knapsack(int i);
    Tw c; //背包容量
    int n; //对象数目
    Tw *w; //对象重量的数组
    Tp *p; //对象收益的数组
    Tw cw; //当前背包的重量
    Tp cp; //当前背包的收益
    int *bestx;
};

template<class Tw, class Tp>
class HeapNode1{
public:
    operator Tw( ) const { return uprofit; }
    bbnode *ptr; //活结点指针
    int level; //活结点所在层
    Tp uprofit; Tw weight; Tp profit;
};

template<class Tw, class Tp>
void AddLiveNode3(MaxHeap<HeapNode1<Tw, Tp>>&H, bbnode *E, Tpup, Tpcp, Tw cw,
bool ch, int lev)
{//向最大堆 H 中增添一个层为 lev，上限重量为 wt 的活结点
//新结点是 E 的一个孩子
//当且仅当新结点是左孩子时，ch 为 true
    bbnode *b=new bbnode; b->parent=E; b->LChild=ch;
    HeapNode1<Tw, Tp> N;
```

```

        N.uprofit=up; N.profit=cp; N.weight=cw; N.level=lev; N.ptr=b; H.Insert(N);
    }

template<class Tw, class Tp>
Knap<Tw, Tp>:: Knap(int cc, int*ww, int *pp, int num)
{   n=num; c=cc; w=ww; p=pp; cw=0; cp=0; }

template<class Tw, class Tp>
Tp Knap<Tw, Tp>:: MaxProfitKnapsack( )
{//返回背包最优装载的收益
    //bestx[i]=1 当且仅当物品 i 属于最优装载
    //使用最大收益分枝定界算法
    //定义一个最多可容纳 1000 个活结点的最大堆
    MaxHeap<HeapNode1<Tp, Tw>>*H;
    bbnode * E;
    H=new MaxHeap<HeapNode1<Tp, Tw> >(1000);
    bestx=new int [n+1]; //为 bestx 分配空间
    //初始化层 1
    int i=1; E=0; cw=cp=0; Tp bestp=0; //目前的最优收益
    Tp up=Bound(1); //在根为 E 的子树中最大可能的收益
    //搜索子集空间树
    while (i!=n+1) { //不是叶子
        //检查左孩子
        Tw wt=cw+w[i];
        if (wt<=c) { //可行的左孩子
            if (cp+p[i]>bestp) bestp=cp+p[i];
            AddLiveNode3(*H, E, up, cp+p[i], cw+w[i], true, i+1); }
        up=Bound(i+1);
        //检查右孩子
        if (up>=bestp) //右孩子有希望
            AddLiveNode3(*H, E, up, cp, cw, false, i+1);
        //取下一个扩展结点
        HeapNode1<Tp, Tw> N;
        H->DeleteMax(N); //不能为空
        E=N.ptr; cw=N.weight; cp=N.profit; up=N.uprofit; i=N.level;
    }
    //沿着从扩展结点 E 到根的路径构造 bestx[ ]
    for (int j=n; j>0; j--) { bestx[j]=扩展>LChild; E=扩展>parent; }
    return cp;
}

```

### 15.2.3 最大完备子图

完备子图问题的解空间树也是一个子集树，故可使用与装箱问题、背包问题相同的最大收益分枝定界方法来求解这种问题，见程序 15-8。



```

class CliqueNode
{
public:
    operator int( ) const{return cn; }
    int cn, //该结点对应的完备子图中的顶点数目
        un, //该结点子树种任意结点对应的完备子图的最大尺寸
        level; //结点在解空间的顶点数目
    bbnode *ptr; //指向结点在解空间树中的位置
};

void AddCliqueNode(MaxHeap<CliqueNode>&H, bbnode *E, int ccn, int uun,
bool ch, int lev)
{//向最大堆 H 中增添一个层为 lev, 上限重量为 wt 的活结点
    //新结点是 E 的一个孩子
    //当且仅当新结点是左孩子时, ch 为 true
    bbnode *b=new bbnode; b->parent=E; b->LChild=ch;
    CliqueNode N; N.cn=ccn; N.un=uun; N.ptr=b;
    N.level=lev; H.Insert(N);
}

template<class VertexType, class EdgeType>
int GraphMatrixND<VertexType, EdgeType>::BBMaxClique(int bestx[ ])
{//寻找一个最大完备子图的最大收益分枝定界程序
    //定义一个最多可容纳 1000 个活结点的最大堆
    MaxHeap<CliqueNode> H(1000);
    //初始化层 1
    bbnode *E=0; //当前的扩展结点为根
    int i=1; //扩展结点的层
    cn=0; //完备子图的大小
    bestn=0; //目前最大完备子图的大小
    //搜索子集空间树
    while (i!=numNodes+1) { //不是叶子
        //在当前完备子图中检查顶点 i 是否与其他顶点相连
        bool OK=true; bbnode *B=E;
        for (int j=i-1; j>0; B=B->parent, j--)
            if (B->LChild && smpEdges[i-1][j-1]==0) { OK=false; break; }
            if (OK) { //左孩子可行
                if (cn+1>bestn) bestn=cn+1;
                AddCliqueNode(H, E, cn+1, cn+numNodes-i+1, true, i+1); }
            if (cn+numNodes-i>bestn) //右孩子有希望
                AddCliqueNode(H, E, cn, cn+numNodes-i, false, i+1);
        //取下一个扩展结点
        CliqueNode N;
        H.DeleteMax(N); //不能为空
    }
}

```

```

        E=N.ptr; cn=N.cn; i=N.level;
    }
    //沿着从 E 到根的路径构造 bestx[ ]
    for (int j=numNodes; j>0; j--) { bestx[j]=扩展>LChild; E=扩展>parent; }
    return bestn;
}

```

解空间树中的结点类型为 **bbnode**，最大优先队列中元素的类型是 **CliqueNode**。当从最大优先队列中选取元素时，选取的是具有最大 **un** 值的元素。函数 **AddCliqueNode** 用于向生成的子树和最大堆中加入结点。函数 **BBMaxClique** 在解空间树中执行最大收益分枝定界搜索，树的根作为初始的扩展结点。**while** 循环不断展开扩展结点直到一个叶结点变成扩展结点，因此最大完备子图已经找到。沿着生成的树中从叶结点到根的路径，即可构造出这个最大完备子图。

## 习题 15

15-1 假定在一个 LIFO 分枝定界搜索中，活结点列表的行为与堆栈相同，请使用这种方法来解决例 15-1 的背包问题。LIFO 分枝定界法与回溯法有何区别？

15-2 对于如下 0-1 背包问题： $n=4$ ， $p=[4, 3, 2, 1]$ ， $w=[1, 2, 3, 4]$ ， $c=6$ ，问答以下问题。

(1) 画出有 4 个对象的背包问题的解空间树。

(2) 像例 15-1 那样，描述用 FIFO 分枝定界法解决上述问题的过程。

(3) 使用程序 14-5 的 **Bound** 函数来计算子树上任一叶结点可能获得的最大收益值，并根据每一步所能得到的最优解对应的定界函数值来判断是否将结点加入活结点列表中。解空间中哪些结点是使用以上机制的 FIFO 分枝定界法产生的？

(4) 像例 15-1 那样，描述用最大收益分枝定界法解决上述问题的过程。

(5) 在最大收益分枝定界法中，若使用 (3) 中的定界函数，将产生解空间树中的哪些结点？

15-3 在程序 15-4 中增加代码，将指向由函数 **AddLiveNode** 生成的结点的指针存储在一个链表队列中。**MaxLoading** 必须利用这些指针信息在程序终止之前删除所有生成的结点。

15-4 在程序 15-6 中，定义一个 **bestw** 来记录目前生成的可行结点所对应的重量的最大值。修改程序 15-6，使得如果活结点的重量大于等于 **bestw**，则将它加入子集树及最大堆中。此外，还必须增加初始化和更新 **bestw** 的代码。

15-5 只使用一个最大优先队列，来实现用最大收益分枝定界法求解货箱装船问题，不要使用程序 15-6 中所用到的部分解空间树，而在每个优先队列的结点中都加入通向根结点的路径信息。

15-6 修改程序 15-6，把删除 **bbnode** 类型和 **HeapNode** 类型结点的任务放在程序结尾处。

15-7 (1) 在程序 15-8 中，若右孩子的 **un** 值大于等于 **bestn**，则将它加入最大堆中。如果将条件设为 **un>bestn**，程序能否正确执行？为什么？(2) 程序是否将 **un $\geq$ bestn** 的左孩子加入最大堆中？(3) 修改程序，使得只将 **un>bestn** 的结点加入到最大堆和生成的解空间树中。

## 参考文献

- [1] 殷人昆. 数据结构(用面向对象方法与 C++ 语言描述)(第 2 版). 北京: 清华大学出版社, 2007.
- [2] 冯俊. 数据结构(第 1 版). 北京: 清华大学出版社, 2007.
- [3] 李春葆. 数据结构习题与解析(A 级). 北京: 清华大学出版社, 2006.
- [4] 唐发根. 数据结构教程(第 2 版). 北京: 航空航天大学出版社, 2005.
- [5] 许卓群, 杨冬青, 唐世渭, 张明. 数据结构与算法. 北京: 高等教育出版业, 2004.
- [6] 侯捷. STL 源码剖析. 武汉: 华中科技大学出版社, 2002.
- [7] 王晓东. 数据结构与算法设计. 北京: 电子工业出版社, 2002.
- [8] Adam Drozdek. Data Structures and Algorithm in C++ Second Edition. 陈曙晖译. 北京: 清华大学出版社, 2003.
- [9] Clifford A, Shaffer. A Practical Introduction to Data Structures and Algorithm Analysis. 张铭, 刘晓丹译. 北京: 电子工业出版社, 2002.
- [10] 陈慧南. 数据结构——使用 C++ 语言描述. 南京: 东南大学出版社, 2001.
- [11] 朱站立. 数据结构——使用 C++ 语言. 西安: 西安电子科技大学出版社, 2001.
- [12] 徐孝凯. 数据结构实用教程(C/C++ 描述). 北京: 清华大学出版社, 1999.
- [13] 严蔚敏, 吴伟民. 数据结构(C 语言版). 北京: 清华大学出版社, 1997.
- [14] 陈小平主. 数据结构. 南京: 南京大学出版社, 1994.
- [15] 黄干平等. 数据结构. 北京: 科学出版社, 1992.
- [16] 窦延平等. 数据结构教程. 上海: 上海交通大学出版社, 1990.
- [17] 拉佛. Java 数据结构和算法(第 2 版). 计晓云译. 北京: 中国电力出版社, 2004.
- [18] 李莲治等. 数据结构. 大连: 大连理工大学出版社, 1989.
- [19] 杨开汉. 数据结构. 北京: 中国财政经济出版社, 1989.
- [20] 罗文化等. 数据结构原理. 上海: 上海科学技术文献出版社, 1988.
- [21] 施伯乐等. 数据结构. 上海: 复旦大学出版社, 1988.
- [22] A V 阿霍等. 数据结构与算法. 北京: 科学出版社, 1987.
- [23] 朱祥华等. 数据结构及应用. 北京: 北京邮电学院出版社, 1989.
- [24] 王颂赞, 费德宝. 数据结构趣谈. 上海: 上海科学技术出版社, 1989.
- [25] 严蔚敏等. 数据结构题集. 北京: 清华大学出版社, 1988.
- [26] 魏晴宇等. 数据结构. 北京: 中国人民大学出版社, 1988.
- [27] 许卓群等. 数据绘声绘色. 北京: 高等教育出版社, 1987.
- [28] 许卓群等. 数据结构. 北京: 高等教育出版社, 1987.
- [29] 计算机专业研究生考试选编. 北京科学技术出版社, 1985.
- [30] 邹海明等. 计算机算法基础. 武汉: 华中理工大学出版社, 1985.
- [31] N 沃思. 算法+数据结构=程序. 北京: 科学出版社, 1984.